Minutes of the 15th ARG Meeting

10-12 February 2002

Cupertino, California USA

Attendees: John Barnes, Randy Brukardt, Gary Dismukes, Robert Duff, Pascal Leroy, Stephen Michell (all but Tuesday afternoon), Tucker Taft.

Observers: Steve Baird (Rational, USA); Richard Riehle (Naval Postgraduate School, USA, Sunday and Monday)

Meeting Summary

The meeting convened on 10 February 2002 at 9:00 hours and adjourned at 15:30 hours on 12 February 2002. The meeting was held in a conference room at the offices of Rational Software Corporation in Cupertino California.

The meeting discussed the cut off of funding for WG9 and ARG activities, and its implications for Ada standardization work. It also discussed the format and development of the amendment document. The meeting covered the entire agenda. The first two days were largely spent on amendment AIs. The third day was largely spent on normal AIs.

By acclamation the meeting thanked the Rational Software Corporation for the facilities and the excellent supply of food and refreshments.

Meeting Minutes

There were no comments on the minutes of the 14th ARG meeting. The minutes were approved by acclamation.

Next Meeting

The next ARG meeting will follow the WG9 meeting at Vienna, Austria on Friday, June 21st " ending on June 23rd. Pascal Leroy will make arrangements.

There was a discussion of the following meeting. The SigAda conference (now planned for December) is too late, given our desire to have three meetings a year, approximately 4 months apart. Tentatively, a meeting around the weekend of the 19th of October is planned, either in Boston or Canada.

Old Action Items

The old action items were reviewed. Following is a list of items completed (other items remain open):

Steve Baird:

- AI-167
- AI-216

John Barnes:

- AI-254
- Working on vector/matrix standard for WG9 (which will be referred to the ARG).

Randy Brukardt:

- AI-85
- AI-246
- AI-248

- AI-260

Editorial corrections only:

- AI-161
- AI-195
- AI-225
- AI-229
- AI-233
- AI-238
- AI-240
- AI-257
- AI-267

Gary Dismukes:

- AI-273
- Investigate the incompatibility impact of the proposed resolution of AI-230.

Bob Duff:

- Fix limited types a bit to permit aggregates and initialization by function call (new AI-287).

Mike Kamrad:

- Amendment on Assert pragma (new AI-286) [Proposal by Tucker Taft].

Pascal Leroy:

- AI-228
- AI-272
- Make a proposal for non-reserved keywords (new AI-284).

Erhard Ploedereder:

- AI-147
- AI-217-04

Tucker Taft:

- AI-230
- AI-251
- AI-266

New Action Items

The combined unfinished old action items and new action items from the meeting are shown below:

Steve Baird:

- AI-216

- AI-251
- AI-280
- Study the recommended level of support in chapter 13 to find any problems with aliased and by-reference types. If any are found, create an AI to correct them.

Randy Brukardt:

- AI-224
- AI-248
- AI-259
- AI-279
- AI-283
- Object_Size attribute

Editorial changes only:

- AI-85
- AI-147
- AI-246
- AI-254
- AI-260
- AI-262

Gary Dismukes:

- AI-158
- AI-196

Bob Duff:

- AI-239
- AI-287 (split into aggregate part and constructor function part)
- Be the test creator of last resort

Mike Kamrad:

- Various items to be standardized [jointly with Mike Yoder]
    - Discard_name & 'image
    - External_tag
    - Storage_IO of tagged types
    - Array indexed by holey enumeration
    - Static elaboration
    - GNAT attributes and pragmas
- CPU time (separate from real-time) [jointly with Joyce]

Pascal Leroy:

- AI-228
- AI-264
- AI-284
- AI-285

Steve Michell:

- AI-148
- AI-250

Erhard Ploedereder:

- AI-237

Tucker Taft:

- AI-133
- AI-162
- AI-167
- AI-188
- AI-214
- AI-217-04
- AI-230
- AI-231
- AI-252
- AI-266
- AI-270
- AI-282
- AI-286 (split into Assert from rest)
- Physical units (length/dimensional analysis), whereby square meters are generated by result of multiplying meters; subtypes with special attributes would be used to provide the specifications of these dimensions, reducing the introductions of lots of extra operators.
- Attempt to find funding for WG9 and the Amendment (separately or together).

Joyce Tokar

- AI-249
- CPU time (separate from real-time) [jointly with Mike Kamrad]

Mike Yoder:

- Various items to be standardized [jointly with Mike Kamrad]
    - Discard_name & ‘image
    - External_tag
    - Storage_IO of tagged types
    - Array indexed by holey enumeration
    - Static elaboration
    - GNAT attributes and pragmas

All:

- Recommend new members.

Funding Issues

Randy reports that the WG9 funding for this fiscal year was abruptly canceled on February 4th. Thus Jim Moore is completely unfunded, and Randy has very limited funding through the ARA.

There was a discussion of the worst case scenario (essentially, no additional funding can be found). In that case, Jim Moore will have to resign as convenor of WG9, and Randy will be able to do only limited work on the Amendment. We'll need a replacement convenor in that case. Steve Michell suggests that it is important to keep the convenor in the US.

The discussion turned to ways to find funding. Tucker believes that customers who depend on Ada for their business should put some pressure on the DoD to fund a minimum level of language maintenance. He will take an action item to contact vendors and customers to try to find funding.

Amendment Document Prototype and Development

Randy explains the format of Amendment Document prototype and his plans for developing the documents. He notes that he had drawn up these plans in the belief that he was going to funded to do this work, so it is likely that not all of these things will be accomplished in the near future.

Randy points out that an Amendment does not have Defect Reports (as a Corrigendum does). Thus, cross-references from the document to AIs are impossible in the final document (AIs are just internal working documents of the ARG). The prototype and later working versions will include such cross-references, but they will be removed from the final version.

The group agrees that drafting the document now will be valuable in organizing further work. Randy notes that he is required to make a matching version of the RM with the changes integrated. He plans to maintain the AARM as well (as he did with the Corrigendum).

Randy asks which items should be included now. He notes that early drafts will be skewed to present only the easier proposals, as those are the only proposals complete enough to include. After discussion, the group agrees that only ARG approved AIs should be included.

Randy asks how the documents should be made available. It is proposed that they are given their own section on the Ada-Auth.org website, with appropriate disclaimers to note that not all proposals being considered are included in the documents. The group agrees with this proposal. The group also decides that these drafts should be announced to the general public for comments.

Detailed Review

The minutes for the detailed review of AIs are divided into existing amendment AIs and non-amendment AIs. The AIs are presented in numeric order, which is not necessarily the order in which they were discussed. Votes are recorded as "for"-"against"-"abstentions". For instance, a vote of 6-1-2 would have six votes for, one vote against, and two abstensions.

Detailed Review of Amendment AIs

AI-216 Unchecked Unions -- Variant Records With No Run-Time Discriminant

Randy and Steve Baird fixed wording from the version distributed in e-mail.

"Any name which denotes…" should be a Legality Rule.

Tucker commented that directions to the implementation should not use "shall". The sentence "All objects of the type shall have the same size" should really be written "All objects of the type have the same size." This is a general principle of the wording in the Ada Reference Manual.

Steve Baird asks if rule: "The type shall not be a by-reference type." is correct. He selected this rule because it is easier than listing all of the items that should be disallowed (tags, finalization, view conversions, and limited types). Tucker wants to allow volatile and tagged types in unchecked unions. The group feels that simply the requirement of the type being C-compatible is enough; the implementation shouldn't define anything to be C-compatible that it can't handle. Steve points out that we can't handle finalization here (because we can't know whether a controlled component exists or not, so we can't implement the as-if semantics), so that has to be disallowed. Therefore, replace the rule with "The type shall not have a controlled part."

The default convention should be C. Steve Baird wonders why it shouldn't be C_Pass_by_Copy. The group prefers to remain consistent with everything else in the language, but of course it can be overridden to have C_Pass_by_Copy if desired.

Someone questions the rule "An unchecked union type is eligible for conventions C and C_Pass_By_Copy, as is any subtype of the type." Tucker reads B.1, and says that it should say C-compatible. A long discussion of "eligible" versus "compatible" ensues. Eventually, the group concludes that "eligible" is what is required, and we need to specify what is required (not what is desirable). So the rule is correct as written: an unchecked union is eligible for both C and C_Pass_By_Copy, and its convention is C by default.

In order to allow these to be C_Pass_by_Copy types, B.3(60.2/1) needs to be changed to allow discriminants.

Pascal comments that pragma Suppress seems to be the wrong model, as it compromises portability. In addition, check failures causes the program to be erroneous. He would prefer a model where check suppression is mandatory; and check failures create abnormal values.

Pascal's change essentially would be to allow these to be used as Unchecked_Conversions. (This is common in C code). We have a straw poll: Should unchecked unions be usable as Unchecked_Conversions? This proposal is defeated 2-4-2.

Tucker wonders if we can get rid of the concept of inferable discriminants? Steve Baird says that it's needed because of the complexity of handling nested discriminants.

Why aren't parenthesized expressions considered inferable discriminants? Steve says this is to match the rules for case statements. This sends several ARG members scrambling to their RMs. Sure enough, a parenthesized case expression requires full coverage. The point is conceded.

Tucker suggesting that getting rid of the need to propagate the discriminant downward would allow us to get rid of this idea. The user could use defaulted discriminants to handle this if they need it. Pascal wonders if removing the capability just to get rid of a few words the manual is worthwhile.

We have a straw poll: “Keep inferable discriminants as in the AI” This vote is inconclusive: 2-2-4.

Approve intent of AI: 7-0-0

Steve Baird will write up changes.

AI-00217-04 Handling mutually recursive types via separate incomplete types with package specifiers

We plunge right into the details of the proposal (and the recent e-mail exchanges on it).

Pascal points out that the Ripple Effect is impossible to handle for the Rational incremental compiler technology. Implicit visibility is not acceptable either (it is very difficult, but not as impossible as a Ripple Effect). The group concludes that you need a with_clause in order to see the completion. This means either an enclosing unit, a **with**, or and inherited **with**. It does *not* mean in the semantic closure. The wording would be something like “Within the scope of the completion or the scope of a with_clause for the completion.”

Tucker writes an example of a question that he has:

```
package P is
    type T is tagged separate in Q;
    procedure Print (X : T);
    type T_Ptr is access all T;
end P;

with P;
procedure Proc (Y : P.T_Ptr) is
begin
   P.Print (Y.all); -- legal?
end Proc;
```

The completion is not visible in Proc; should this call be legal? Tucker argues that it should be: we know (without seeing the completion) that P.T is a by-reference type, and Y is actually a reference, so there is no difficulty in passing it to P.Print.Tucker proposes that for a tagged incomplete formal parameter, we should allow tagged incomplete as an actual parameter. A straw vote on this proposal passes 7-0-2.

Tucker continues with the example:

```
with P;
procedure Proc2 (Y : P.T_Ptr; Z : P.T_Ptr) is
begin
   Z.all := Y.all; -- legal?? – No, requires with Q.
end Proc2;
```

This requires the package containing the completion to be withed or directly in scope. That's because details of the full type are needed to perform the assignment.

```
with Q;
package R is
  A : Q.T;
end;

with P, R;
procedure Proc3 (Y : P.T_Ptr) is
    W : P.T renames Y.all;
begin
   R.A := Y.all; -- legal?? (Tucker would like this be to legal.)
end Proc3;
```

Tucker would like this to be legal, because R.A has the type of the completion, and thus details of the completion must be available. So there is no implementation reason for it to be illegal. This discussion effectively brings up the issue of what the subtype matching rules should be.

The term "completion is available" is defined to mean "in scope of **with** of package containing the completion or immediate scope of the completion."

Steve Baird proposes the rule: "When the completion is available, then the incomplete type is just a subtype of the full type." This would take care of the subtype matching rules. For example:

```
with Q;
package R2 is
  A : Q.T;
  procedure Store (I : out Q.T; J : P.T);
end R2;

with P, R2;
procedure Proc4 (Y, Z : P.T_Ptr) is
begin
   R2.Store (Z.all, Y.all);
end Proc4;
```

Bob Duff points out that this would reintroduce the dreaded Ripple Effect – because adding or removing a with_clause would change the legality of the program. For example:

```
with P;
package R3 is
   type T2_Ptr is access P.T;
end R3;

with R3;
procedure Proc5 is
   A, B : R3.T2_Ptr;
begin
   A.all := B.all;
         -- Legality of this would change if with Q added to R3.
end Proc5;
```

So Steve's rule is discarded. In effect we cannot have legality rules that talk about the completion being available, because that predicate causes Ripple Effects.

A different rule is proposed: "If you have a P.T and Q.T in a context, they match if Q.T is indeed the completion of P.T." Note that the "in" package name must not be a renaming. Of course, P.T matches P.T for the subtype matching rules.

Another issue is considered (again extending the same example),involving two incomplete types:

```
package P2 is
    type T is tagged separate in Q;
end P2;

with P, P2;
procedure Proc6 is
   A : P.T; B : P2.T; -- Are these the same?
   ...
```

After discussion, the group concludes that P.T and P2.T should match.

Steve Michell comments that he would like to be able to change the name of the type. In a large project, you may need to be able to change the names of the things when integrating (in order to meet subsystem specifications).

He proposes the syntax should be:

```
type T is tagged separate Q.T;
```

Tucker notes that if both types are named T ('Object' is commonly used), you can't declare the both unless you can change the name. But you can use a nested package to work around the limitation as noted below:

```
package P3 is

    package Q_Types is
        type Object is tagged separate in Q;
    end Q_Types;

    package R_Types is
        type Object is tagged separate in R;
    end R_Types;

end P3;
```

Perhaps changing the name could be optional? The syntax

```
type <Id> is tagged separate [<Old_Name>] in <Package_Id>;
```

is suggested. A straw poll is taken on this issue:

- Must specify the name of the type (as opposed to some other choice) is defeated by a vote of 1-6-2.
- Support specifying the name (optionally) is defeated by a vote of 2-5-2.

Summarizing the results of this discussion:

1) "completion available" means "in scope of with or immediate scope of completion". If the incomplete type would be illegal in a particular context – check if the completion is available (as described above).

2) Tagged incomplete allowed as actual or formal parameter; dereferencing an access-to-tagged-incomplete allowed as actual parameter (and as a parenthesized actual parameter).

3) Syntax as proposed by Erhard; simple name of incomplete type and simple name of completion are always identical.

4) Subtype matching rules: incomplete matches incomplete if same full name of completion; incomplete matches complete if complete has the full name specified as the completion (taking renames into account).

5) No post-compilation rule.

Approve intent of AI: 7-0-0

Tucker will write the wording as soon as possible.

AI-230: Generalized use anonymous access types

Tucker describes the changes to the AI. One of the problems of having a variable of an anonymous access type is that you can view it as having a longer lifetime. It is OK to reference something that lives longer, but not shorter. So, he didn't want to have any variables of an anonymous access type with an object's level.

So, for components that aren't discriminants, they have the same level as if a named access type was declared at the same level. Discriminants of a non-limited type are the same; limited type (which cannot be altered) can take their level from the object. Similarly, for **in** parameters, constants, and other contexts that can't be modified.

Also, should we allow this for function returns? Tucker notes that this seems to solve a problem similar to the one solved by limited function returns. So this doesn't seem to be worth the effort; drop the function return things.

The equality operator is weird, but it seems to be needed. It is suggested to live in a package of its own: Ada.Access_Equalities. Another suggestion is to make an object attribute 'Is_Null which indicates whether or not its prefix is null. This gains support: it would actually be useful in contexts other than anonymous access types: there are many **use type** clauses on access types out there just to test for **null**.

We have a straw vote on 'Is_Null replacing the equality operator. This is adopted overwhelmingly: 8-0-1.

This would make it hard to compare two values. To do it, you would need a local named access type, and then A.all'Access = B.all'Access (or Local_Ptr(A) = Local_Ptr(B). This is not common enough to worry about.

Do we need anonymous allocators? Yes, and they are no worse than the ones already in the language. (Tucker admits those were a mistake, but we're stuck with them now.)

Approve intent of AI: 6-0-1

Tucker will write wording.

AI-248: Directory Operations

Summary: "…is [proposed to] provide{s} portable…" Randy is directed to write in the software present tense.

Rename Extension_Name to Extension.

Someone wonders how the package would deal with the version part of a VMS name. Several ways to do it were suggested. We don't really want to worry about standardizing that.

Tucker wants Base_Name to be the root portion of the file name. Randy explains that combining a base name plus extension to create a simple name. The group agrees that that is a correct model.

But then, how do we create a simple name? Compose is defined to return a full name. The group thinks that Compose should just be a string manipulation function. Thus, it should be able to return a simple name, or some other name. So, it does not return a full name and it does not use the current default directory. The middle parameter Name cannot be null, so there should not be a default on Name, and there should be a check.

Steve Michell would like to be able to easily find all items in a search. It is suggested that there should be an implementation defined string constant ("*" for Unix/Windows) that matches everything. An alternative would be to have the null pattern conventionally have this meaning. Consensus is to use the empty string.

Steve Michell would like additional functionality to be able to access the creation time of files. Randy notes that there is advice to create child units to support system-specific functionality. Steve claims that creation time is widely available. Other people disagree. There is no support for Steve's idea.

Randy asks whether Containing_Directory should add the current directory. Tucker would prefer that Containing_Directory is a purely string manipulation function; use Full_Name to add the current directory. The group agrees, because Full_Name can be expensive.

Tucker would like "file name" to not include directory names (in part because the parenthetical remarks to remind casual readers of that are annoying), but others disagree. The group concludes to leave this defined as it is in the draft AI.

Fix the incorrect name for Set_Directory in the description of Current_Directory noted by Nick Robert's e-mail.

Tucker wants a Form function. Tucker claims that the form information can be determined by querying the system. Randy and others disagree. If it cannot be obtained by querying the system, it would be difficult to implement, because there isn't a directory object (as there is a File_Type object) in which to store this information. After further discussion, this function was decided to be unnecessary and the idea was dropped.

The question of whether a Form parameter should be added to Copy_File is discussed. It seems that it should be added for consistency; it would be necessary to specify permissions on the resulting file, for instance. A straw poll is taken: the vote is 5-0-4.

The group decided to look at the complete list of Nick Robert's comments (see the appendix). Many items were quickly considered out of scope for this proposal.

Locking is too hard to do on Unix, so we can't require it. The group shows no interest in changing names as proposed.

No one supported the idea of a Temporary_Directory. Tucker suggests a Temporary_File_Name function as an alternative. We have a straw poll on the idea: the vote is an inconclusive 3-3-3.

Root_Directory gets an endorsement from Pascal. But Tucker wonders how this would be used. No one has an answer. No further support is found for the function.

The idea of adding Rename to Text_IO and the other predefined IO libraries was considered too much change for too little value. Rename_Files can be used for this.

Someone suggests adding "Exists" to the file name properties. Why should it be necessary to open a file in order to find out if it exists? The idea gets general support.

Someone again objects to the Is_Valid function for search type. Randy explains that it is used to determine when we've run out of items. A More_Entries function could be used instead, but it would be the same amount of work to use.

John asks for an example of a search. Randy fails to note that such an example already exists in the AI's !example section. So he laboriously writes another one on the whiteboard:

```
S : Search_Type;
D : Directory_Entry_Type;

...

Start_Search (S, ...);
loop
    Get_Next_Match (S, D);
    exit when not Is_Valid (D);
      ... -- Process D.
end loop;
```

Then, he goes on to show the alternative:

```
Start_Search (S, ...);
while More_Entries (S) loop
    Get_Next_Match (S, D);
      ...-- Process D
end loop;
```

Several people note that they rarely use **while** loops. But of course More_Entries can be used in an **exit**.

We have a straw poll. Use More_Entries instead of Is_Valid for Search_Type passes 7-0-2. In this case Get_Next_Match raises an exception if nothing is available. A second straw poll asks whether we should remove Is_Valid for Directory_Entry_Type. This also passes 6-2-1.

A discussion of the naming of More_Entries ensues. The names More, Is_More, and More_Entries are suggested. More_Entries is the final selection. Change Get_Next_Match to Get_Next_Entry to be consistent.

Approve intent of AI: 7-0-0.

AI-251 Abstract Interfaces to provide Multiple Inheritance

Tucker explains what's new in the AI: **interface** is an unreserved keyword; private extensions may not have private interfaces; **is null** is now a first class part of the proposal.

Steve Baird points out that this proposal causes problems for compilers that allow the tag to be positioned in the type (using representation clauses). Rational had customers who needed this capability. For example:

```
type T1 is tagged null record;
for T1 use record
    Tag at 92 range ...;
    ...
end record;

type T2 is tagged null record;
for T2 use record
    Tag at 42 range ...;
    ...
end record;

type NT1 is T1 and Interf ...

type NT2 is T2 and Interf ...
```

Now, how does Interf'Class find the tag (because it is in different locations)?

Tucker claims that this can be solved with a tag fetcher in the interface table. A fat pointer implementation would be required in this case.

Approve intent of the AI: 6-0-1

Steve Baird will write words.

AI-254: Downward closures for access to subprogram types

John explains that he did what the minutes said.

Anonymous access to protected subprograms – we don't want them. Delete the "(Are we happy with this?)".

Need a runtime, not compile-time check for null.

The convention of an access parameter is the convention of the subprogram of which it is a parameter.

Gary comments that he really hates this proposal. He does not believe there is enough justification for adding this capability. Several people point out that this probably is because GNAT supports 'Unrestricted_Access for this and it is widely used. This is a portable solution that would work in all compilers.

Pascal notes that this is a new kind of type in the language, so this is a big change to compilers.

Approve AI with changes: 5-0-2

The group agrees that this is the best and simplest solution to the problem. We are not as sure that it has sufficient value to include in the Amendment.

Should we send it to WG9 and include it in the Amendment document now? A straw vote demonstrates that the group would rather wait: 1-3-3.

Therefore, this AI will be put into limbo for later reconsideration.

AI-260: How to control the tag representation in a stream

Correct penultimate sentence of problem: “being [be] sent.”

It turns out that the wording is much easier to understand than the proposal. Leave only the first paragraph of the proposal, then refer to wording for the rest.

Under wording, correct: S’Tag_Read → S’Class’Tag_Read

There is a suggestion to delete the paragraph “User-specified Tag_Read and Tag_Write attributes may raise an exception if presented with a tag value not in S'Class.”, since it is obvious. Tucker would prefer this to be a note, using “should”. After discussion, the group agrees.

John wants “reading).” corrected be “reading.)”

Approve AI with editorial changes: 7-0-0

AI-262: Private with

The changes made (from the Leuven meeting) are discussed.

Tuck asks if an inherited public **with** overrides an explicit private **with**. Yes, the most visibility is the correct result: you can’t take any visibility away this way.

There is some discussion as to what these do on subprograms. For a subprogram body without a separate specification, perhaps a private with should not make things visible in the public specification.

That seems too complicated; it is suggested that private **with**s are allowed only on package specifications or generic package specifications.

The processing of these in an implementation is similar to processing for private units, which become visible at the keyword **private**.

Can we use a private **with**ed unit in a context **use** clause? No, that causes interesting problems. It would make it necessary to check the legality of any use-visible declaration, not just compilation units. Private **with**ed units can be used in pragma Elaborate, etc.

Tucker is concerned about the wording of the legality rule, because it must not traverse package renamings. Change the rule to:

> In the visible part of a package or generic package, a name shall not denote a declaration mentioned only in a with_clause which includes the reserved word **private**.

Approve AI with changes: 7-0-0

AI-266: Task termination procedure

Pascal wonders if we need “local task group”. Tucker says that we need some mechanism for specifying the task group of child tasks.

Tucker muses that he would prefer to revise the proposal to separately specify the task group that child tasks are created with, rather than changing the task group of the current task.

The initial task group does nothing at all; all of its operations have null bodies. Randy comments that that might as well be specified as the default behavior. It is essentially the same as the current semantics of Ada. The AI should explain that the null task group is used for the environment task. It seems that we also need a mechanism to reset the task group to null.

Steve Michell wonders why the proposal does not have any query functions. They are not needed; the intent that is that handlers will be written for these operations to receive notifications.

The proposal should give permission to allow another task to call the Aborted operation, so as to not burden implementation models.

A suggestion is made to add an operation to notify when a task joins or leaves a task group. If you want to iterate through all of the members of a task group, such an operation would allow keeping a list. This idea has general support.

Tucker is not sure whether we still need to be able to change the task group of a running task. If not, removing that capability could simplify the proposal. In particular, we wouldn't need an operation to notify about leaving a task group, because the only way to leave would be via termination.

Steve Michell would like to be able to put top level tasks into different groups. Tucker says that can be done, by changing the local task group appropriately. Steve is convinced.

The description of Local_Task_Group fails to say that it is the group that is used when tasks are created.

Someone suggests that perhaps the reason for the lack of enthusiasm for local task groups is the name. Perhaps the name ought to be changed. Bob suggests “Within_Group”. There is even less enthusiasm for this suggestion.

What happens if you allocate a Local_Task_Group or use arrays of Local_Task_Groups? Do they have to work in a last-in, first-out fashion? Probably make it a bounded error to use Local_Task_Groups in this fashion.

John suggests that we need to keep this proposal as simple as possible consistent with its objectives.

Gary asks about the overhead of this feature? It appears to be a distributed overhead. Yes, but it is on task creation and termination (which are already expensive operations).

A way to mitigate the distributed overhead is to have a restriction No_Task_Groups. If No_Task_Groups is given, there cannot be a semantic dependence on Ada.Task_Groups. Such a restriction should be added.

The proposal originally came from Ravenscar, but they wanted something simpler (which became restriction No_Task_Termination). Then, the Exception Workshop at Ada Europe 2001 identified a similar need; this proposal was created to address their needs. Randy should update the appendix to say this—the appendix implies that this is somehow related to Ravenscar.

We return to the “add a task to a group” operation. When should it be called? When the task is elaborated, or when it is activated, or when it is created? The task group is added when the task is created, so that is when the “add a task to a group” operation should be called. It should be called by the creator. Certainly, it should be called before Never_Activated.

Steve Michell would like a routine that is called at the point of telling the parent that the activation is successful (at the **begin**). He withdraws suggestion after discussion.

Tucker should write the AI before April, and Steve Michell and Joyce will bring it to the HRG.

Approve intent of the AI: 5-0-2

AI-273: Use of PCS should not be normative

Gary describes the AI. The discussion at last meeting was to do whatever you want. But try to be similar to the existing PCS.

Steve Michell suggests that the wording should say “alternative declarations” (plural), so as to allow an implementation to support more than one PCS. The group decided that this kind of permission is useless, as an implementation is always allowed to provide alternate units using different libraries/flags/whatever, and the language should not discuss how this is done.

Approve AI as written: 7-0-0

AI-278: Task entry without accept statement

Several people are against this change. This change provides no help for families, and is (mildly) incompatible. Implementations can always emit a warning if they are able to detect such a situation.

The AI is voted “No action”: 7-0-0

AI-281: Representation of enumeration type image attribute

Tucker suggests that a user defined Image function might be useful, but the proposed capability isn’t valuable itself. We discuss a user-defined Image function. This would have to have composition rules, would have to hook into Text_IO (for Enumeration_IO.Put), would need a corresponding Value function, a lexer for Enumeration.Get, and probably more. This seems too complex for the benefit.

The AI is voted “No action”: 7-0-0

AI-282: Ada unit information symbols

Rational has problems with this proposal, they don’t have line numbers to use in general, because of incremental compilation. They have something known as “source position” to use in the debugger.

If we do this, it is suggested that it be in terms of calls, not in terms of the location of the use of the attribute. Here is a way the feature could be used:

```
procedure Debug (B : Boolean) is
begin
    if not B then
         Put_Line(Debug’Call_Position & “failure”);
    end if;
end Debug;
```

This procedure could then be used all over the application.

Bob suggests something like Exception_Occurrence, with an associated package:

```
Put_Line (Some_Stuff(Debug’Caller_Info) & “failure”);
```

Perhaps this should be named ‘Call_Stack.

Perhaps this should link to exception information.

The group votes to keep this AI alive to work on this idea (abandoning the existing text): 5-0-3

Tucker will chat with Bob and keep the AI alive.

AI-284: Nonreserved words.

Tucker would prefer that we call them keywords. Then change every occurrence of reserved word to keyword; only 2.9 would discuss the differences between reserved words and other keywords.

Bob Duff indicates that the term “reserved word” is common in the RM. After some fiddling with tools, he produces a count of 96 occurrences of the term “reserved word” in the AARM.

Pascal notes that this is too many occurrences to change. Randy concurs, noting that every such paragraph would be marked as a change in the Amendment document and in the consolidated RM.

However, the idea to call them keywords is accepted. Tucker suggests the age-old solution of defining “reserved word” to be equivalent to “reserved keyword”. In the more general case, we simply say “keyword”. This should restrict the changes to 2.9.

Note that this AI doesn't actually define any keywords; they belong in the appropriate AIs where they are used.

Lower difficulty to Medium.

We have a straw vote: Is this a useful capability? This passes unanimously: 7-0-0.

During the discussion, John mentions that he likes the notion of using keywords (as opposed to pragmas) for dealing with the overriding vs. overloading issue addressed by AI95-00218. Maybe this issue should be revisited.

Pascal will work on the final AI.

AI-285: Latin-9 and Ada.Characters.Handling

Bob says that his understanding is that Ada should have tracked the evolution of the ISO standards related to character types. That probably includes support for 32-bit characters (Wide_Wide_Character?) and for Latin-2 to 9. Of course this is a big can of worms, and it is unclear if there is a lot of user demand. But at a minimum, the language should not get in the way of supporting additional character sets.

Pascal will write the AI, as he is most interested in it (support for the additional French characters).

AI-286: Assert Pragmas and Invariants

Pascal is concerned that suppression of assertions could change program behavior. That could happen if an assertion has side-effects. He would like the assertions to always run, just no check.

Tucker thinks that would be a strange semantics. We can’t eliminate side-effects (there is no way to write a reasonable rule to do so), but we should discourage them. Gary notes that GNAT suppresses the entire expression evaluation when assertions are turned off.

Someone asks what this pragma buys us over ordinary code controlled with a Boolean constant. The answer is that these pragmas can be placed in a declarative part (without contortions).

Another question is whether these expressions can be used to provide information to the optimizer. That is, when the assertion is suppressed, can the optimizer assume that it evaluates to True? (It certainly can do that when it is not suppressed.)

Tucker asks to suppress the suppress discussion, and discuss merits of proposal itself. (We'll come back to the suppress discussion on Tuesday.)

So we turn to pragma Assert. Must the string expression be static? No. The string expression is only evaluated if the assertion expression failed. We want to add a special permission so that Assertion_Error can rename something else for compatibility with existing implementations. (This is needed so that the Exception_Name would be allowed to return something else other than Assertion_Error).

We now turn to the pre and post conditions for a subprogram.

Tucker explains the basics of the proposal. These pragmas are on the specification of the subprogram; they're not allowed in the body. The expression is in the context of the subprogram. Pascal claims this is hard to implement. Tucker responds that it is similar in visibility to a record representation clause. But it might be hard to implement

record representation clauses in a particular compiler, and at any rate the mechanism used for record representation clauses may be somewhat ad-hoc.

The value of the proposal is that these expressions are visible to the specification, and thus can be used to prove things about calls. They are more valuable than comments, because these are known to be legal Ada (the compiler has checked them) and may be easier to read than English. John thinks that the last point is arguable: a sentence like "the matrix shall not be singular" conveys a lot of information.

Steve Baird thinks that these pragmas are (also) necessary in the body. Tucker replies that use inside of a body is the same as pragma Assert.

Pascal expresses concern about bloating the specification of packages (making it harder to find things). Is this really useful? Various attendees express support for this feature. Most like having these available in the specification.

Next, we discuss Inherited_Preconditions and Postconditions for subprograms. These are inherited as needed. What are the pre and post conditions for a dispatching call? Tucker explains the rules by showing an example:

```
procedure Update (X : T);
pragma Inherited_Precondition (Foo (X));
pragma Inherited_Postcondition (Bar (X));

Update (<T'Class>);
```

All conditions of the actual subprogram are used. For postconditions, all of them are "and"ed together. So you can assume that the Inherited_Postcondition is True.

For preconditions, all of them are "or"ed together. This means that weakening of preconditions is allowed.

John wonders if we need a way to reference the parent's precondition? Tucker doesn't think that is necessary.

Someone suggests that preconditions are "or"ed in; they can be overridden by respecifying Inherited_Precondition.

Steve Michell tries a concrete example:

```
generic
    type Item is private;
package Stacks is

   type Stack is abstract tagged private;

   function Is_Full (S : Stack) is abstract;
   function Is_Empty (S : Stack) is abstract;
   function Top (S : Stack) return Item is abstract;

   procedure Push (S : in out Stack; I: Item) is abstract;
   pragma Inherited_Precondition (Push, not Is_Full (S));
   pragma Inherited_Postcondition (Push,
               not Is_Empty (S) and Top(S) = Item);

   procedure Pop (S : in out Stack; I : out Item) is abstract;

end Stacks;

generic
package Stacks.Array_Stacks is
    type Array_Stack (Len : Positive) is new Stack with private;

    function Current_Size (S : Array_Stack) return Positive;

    procedure Push (S : in out Array_Stack; I: Item);
    pragma Postcondition (Push, Current_Size(S) > 0);
```

```
end Stacks.Array_Stacks;
```

Are calls in expressions of these pragmas dispatching? Not necessarily, we don't want special rules in these expressions. Long discussion about dispatching between Bob and Tucker.

We discuss whether these ought to be restricted to pure functions. There is quite a bit of support for that idea.

Richard's opinion on the basic proposal: he has a project (at the Naval Postgraduate School) which is using Eiffel instead of Ada because of preconditions/postconditions. He doesn't think that these would be a debugging tool per se, but they may be turned off in a deployed system. Seems to be a lot of customers that would find these helpful.

Pascal suggests splitting this AI into an Assert pragma part, and the rest of it.

The question comes up as to user demand. Vendors don't see demand for this from users. That doesn't seem to be a useful criterion for evaluating amendments. Users are unlikely to ask vendors for a large new feature that impacts many parts of a compiler, or to be able to design a large new feature sufficiently.

Invariants are very important because they can't be written in Ada 95. (While there are ways to do preconditions, postconditions, and the effect of Assert.)

So the discussion turns to invariants.

```
pragma Package_Invariant (Package, Boolean_Function_With_No_Params,
                          [Message=>] String);

pragma Type_Invariant (Type, Boolean_Function_With_One_Param,
                       [Message =>] String);
```

These are postconditions. These should apply to all visible operations on every call. (That is a change from the AI as written.) The reason for this change is that defining "external call" would be difficult (considering generics and inlining). Another way to say this is that it applies to anything declared in the visible part.

Does an invariant for a derived type apply to inherited subprograms that are not overridden? Tuck thinks that it should not. If it did apply, calling the parent operation directly would be semantically different from calling an inherited one. Pascal agrees with Tucker, because it would be error prone; the invariant would apply only if the operation is overridden. For example, consider a "+" inherited for an integer "prime number" type. The conclusion is that conversions, including view conversions, do check the invariant, so the effect is that the derived type invariant is checked.

These are inherited for public children.

It is necessary that invariants are enforced on conversions. This must happen any time there is a conversion, including implicit ones. (Note that this means that something expensive can happen for implicit conversion.)

Do invariants apply to the private part? No, invariants apply to external interfaces; within the subsystem, the invariants can be violated. Possibly private parts could have their own invariant (for calls from children).

Someone suggests that perhaps a subtype invariant would be useful. It would happen wherever a "belongs to subtype" check happens. This would be more similar to what the language already provides for constraint checks, but it would only make sense for scalars, discriminants, bounds, and stuff that can only be assigned by whole-object assignment. Otherwise the invariant can be violated by assignment to a part of an object.

Another suggestion is made: allow a type invariant only on a private type and extensions. That's because changing components directly would not be checked; thus it would be of a dubious value on non-private types.

A long discussion ensues about the difference between Type_Invariant and Subtype_Invariant. The conclusion is that the primary difference is the point of the check; there is no other real difference.

Steve Michell asks what would be checked in the following example:

```
type Bar is
    record
        Len : Integer;
        Data : String_Access;
    end record;
pragma Subtype_Invariant (Bar, (Len /= 0) = (Data /= null));
-- Or maybe Type_Invariant.

X : Bar;

...

X.Len := 7;
X.Data := new String'("Hello..");
```

The answer is that there would be no check here. You need abstractions in order to have useful Type_Invariants.

Discussion slowly shifts toward Tucker's position. Subtype_Invariants make most sense for discriminants and elementary types and for bounds. That's because those items can change only when the subtype check is applied.

Type invariants are allowed only on new declarations.

```
package P1 is
    type T1 is tagged private;
private
    type T1 is
        record
            X, Y : ....;
        end record;
end P1;

package P1.P2 is
    type T2 is new T1 with private;
private
    type T2 is new T1 with
        record
            Z : ...;
        end record;
end P1.P2;
```

Does the invariant apply to the view conversion downward (for an in out parameter)? Yes.

We take a straw poll: Assert is useful and should be standardized. This passes easily: 9-0-0.

We conclude that it should be split off into a separate AI.

Another straw poll: Should we continue work on invariants/preconditions/postconditions? This passes: 6-3-0.

Most of the group agrees that invariants are the most useful. (More so than the pre and post conditions).

On Tuesday, we take this AI up again.

Tucker starts a discussion of the suppress semantics for Assert. He lists the possibilities:

1. Turned on;
2. Ignored completely; (ignore pragmas completely, can't use information in them to optimize)
3. "Suppress" semantics (may assume true)
    a. No evaluation of Boolean expressions (if false, it is erroneous)

b. Always evaluation of Boolean expressions

c. May have side effects (might evaluate the Boolean expressions)

4. Assume true if no side effects possible or you do the check.

Someone asks, in case (2), do expressions in these pragmas cause freezing? Are they even analyzed for legality? Probably they are analyzed and should freeze.

We clearly need case (1).

We ask the vendors present about the existing semantics of this pragma. GNAT defines that turning off Asserts still affects legality, but does not evaluate the expression. So this is case (2) (the expression doesn't get expanded in the back end, so it has no effect on the optimizer). Apex does not allow any sort of suppression. Pascal says that some sort of suppression is necessary, they just haven't figured out what kind.

A suggestion is made to have several check names for these cases. One suggestion is:

- Suppress(Assertion_Checks); — case 3b
- Suppress (Assertion_Eval); — case 2

Bob Duff wants case (3a) supported. The suggestion is revised to:

- Suppress (Assertion_Pragmas); — case 2
- Suppress (Assertion_Evaluation); — case 3a
- Suppress (Assertion_Checks); — case 3b

Which of these three is meant by Suppress (All_Checks)? Generally, we want the fastest code in this case, so it should mean Assertion_Evaluation.

There still is discomfort about Suppress here. Multiple check names for the same check don't help. Perhaps it is better to avoid Suppress altogether, and use a new pragma. Pragma Assertion_Policy is proposed:

```
pragma Assertion_Policy (policy_identifier);
```

Where the legal policy identifiers are:

- Check — case 1;
- Ignore — case 2;
- Evaluate — case 3b;
- Assume_True — case 3a

The default policy is Check. This is a configuration pragma. It doesn't need to be consistent over an entire partition (that is, different units may have different policies).

How does GNAT suppress checks? It suppresses with a compiler switch (actually, enables with a switch). So it doesn't have a precedent for a pragma.

Can we give permission to add additional policies? Sure.

We have a straw vote on this pragma: it passes 7-0-0.

After a break, Tucker shows the following chart regarding the rules for pre- and post-conditions:

- Precondition Not inherited.
- Classwide_Precondition "or"ed in, controlling operands of type T'class
- Postcondition Not inherited.
- Classwide_Postcondition "and"ed in, controlling operands of type T'class
- Type_Invariant Not inherited.
- Classwide_Type_Invariant "and"ed in, controlling operands of type T'class
- Package_Invariant

Classwide_Precondition is used to specify the things that can be handled; Classwide_Postcondition is the guarantee on the result.

Again, Tucker makes the point that Invariants are like postconditions.

Gary asks for a worked out, semi-real world example of the use of this feature.

AI-287 Limited types considered limited

Based on the e-mail discussions, Tucker suggests splitting this into two AIs, the easy piece (limited aggregates), and the constructor functions. Pascal worries that it would be rather ugly to do only aggregates, when functions are used much more often for initialization.

For aggregates, the proposal is essentially that limited aggregates are "built in place", and to allow subtype names in place of component expressions.

"Built in place" is already required by AI95-00083 in TC1, so this is not a new mechanism for compilers.

Randy wonders if there is a resolution problem with allowing subtype names to appear in the context of expressions. The answer is no, because you already know the type of the component.

Steve Baird notes that you can write "Integer" for a component, so you get an uninitialized aggregate component. Can you raise an exception in this case? No, it is not default initialization; it is just "uninitialized". So the use of a subtype name in a aggregate would *not* be equivalent to declaring an object and assigning it, because there is no assignment, it's all built in place.

Bob notes that there must be a check that the subtype is compatible with the component's constraint (as in 3.2.2(11)). Tucker suggests that the rule ought to be the same as a default initialized allocator (4.8(9-10)).

Gary points out that we can't allow giving a subtype_mark for a discriminant, as that would lead to an uninitialized discriminant. The group agrees.

Limited aggregates would be allowed as initializers, and passed as parameters.

A brief discussion of the value of this change occurs. Limited types occur often, and the use of them is often problematic. Most ARG members share a war story about limited types at this point. The general agreement is that fixing this is cheap and useful.

We have a straw poll on the aggregate proposal as written, with two changes (subtype compatibility check, and no subtype_marks for discriminants): it passes easily, 9-0-0.

The discussion turns to constructor functions.

Several members express interest in Dan Eiler's proposal of a procedure renamed as a function (allowing a procedure to be called with function syntax). Tucker says this doesn't work because the **out** parameter would have to be default initialized.

Tucker's latest proposal is that the caller always would allocate space for a limited return type, possibly using a pragma to specify that this doesn't need to happen. Steve Baird notes that the function has to indicate whether or not the passed in buffer is used. So the model is that the caller allocates some object, and the function returns a bit indicating whether or not the allocated object was used.

Randy wonders how this would work with unconstrained array objects that are limited. It is suggested that they be illegal, but that is incompatible with Ada 95.

Tucker discusses the possible ways that a function could work:

```
function Blah (...) return Lim_T is  --
```

- Return a reference to an existing object (or existing collection).
- Return a newly constructed object:
  - Create the object in the return statement itself;
  - Or "declare" and initialize a view of the object in the caller's space.

To handle the last case, we need some new syntax to designate the object allocated in the caller's space. As usual when it comes to syntax, the ARG members demonstrate their unlimited creativity:

```
Blah'Result : Lim_T := …;

Result : out Lim_T := …;

Result : return Lim_T := …;

function Blah (…) return
    X : Lim_T := … is
    …
begin
    …
```

Once this new syntax is used, the "magic" object must be returned; it must be declared in the outmost block; the return syntax could be "return;" (in which case falling off the end would be OK), but it seems that it is better to explicitly return the "magic" object, for readability.

This capability must be supported for all types, because of the contract model. If the type is not a return-by-reference type, then the result can be a copy (so no code generation change is needed).

Steve Baird notes that in the case that the actual is more constrained than the formal, Constraint_Error could happen.

Randy still doesn't understand how a limited unconstrained array object could be handled this way. He gets no answer.

For functions, the object is allocated by the caller, and initialized by the called function.

Pascal wonders why this is different than a newly constructed object. Tucker replies that you can't return a newly constructed object.

Randy and Steve Baird wonder if you need to pass in masters (for task components) and finalization pointers (for controlled components). Tucker replies that it may be necessary in some implementation models, but "current" master should work for most implementations.

This proposal would cause a performance hit to return-by-reference functions compared to Ada 95. Some people would consider this an incompatibility. One way to avoid this would be to indicate this semantics in the function profile (Bob's original idea). However, that would have to appear anywhere the profile of a function can occur (including access to functions and generic formal function). This seems like too big a change.

We take a series of straw polls.

The question of whether we should explore this idea further passes 5-0-3.

Should we use new syntax visible in the function specification, or Tucker's proposal that this be the default behavior. Tucker's proposal gets most of the support: 2-5-1.

Do we want some mechanism in the function body to name the result object? This also passes 4-2-2.

More straw polls are taken:

Syntax: should the keyword **return** be used for declaring the return object (third syntax above)? Passes easily: 5-1-2.

Should procedure style return statements be used? Rejected: 2-5-1

Steve Baird is concerned about exceptions raised in aggregates/return statements, where does finalization occur in that case?

Bob will write both AIs up.

User-defined assignment checks

Tucker makes a proposal. If a predefined check for an assignment statement fails (that is, would raise Constraint_Error), then a user-defined routine is called. This is the other half of the user-defined assignment problem.

There is not much support for this idea, no AI is assigned.

Detailed Review of Regular AIs

AI-85: Append_File, Reset and positioning

Change the !subject to include “Stream_IO”, because that is what we’re talking about.

!response, bullet 3, “Thus, [A]{a}n implementation…” “makes sense” → “is appropriate for”

Approve AI with changes: 6-0-1

AI-147: Optimization of controlled types

Tucker complains about the effect of Initialize; the minutes of the previous meeting say that this should simply be user-defined. The group agrees, change bullet (a) in the wording and summary to say “If the Initialize procedure is not user-defined, and…”.

Bullet (b) should use the same wording as 11.6(5) to describe the external effect: “some effect on the external interactions of the program”.

Simplify the summary. End at “is extended.”

Approve with changes: 7-0-0

AI-167: Unchecked_Conversion is never erroneous

Steve Baird explains the AI. He proposes a change: adding a defaulted Boolean to Unchecked_Conversion to specify whether or not a validity check is made. This would look like:

```
generic
     type source (<>) is limited private;
     type target (<>) is limited private;
     Validate : Boolean := False;
function Ada.Unchecked_Conversion (S : Source) return Target;
```

Someone asks what this does for non-elementary types (which don't have a 'Valid operation). The answer is that is implementation defined. People are unconvinced, and compatibility worries are expressed (even though instantiations are completely compatible). Tuck would prefer a new generic.

Richard says he uses:

```
if Convert(X)’Valid then
   Y := Convert(X);
end if;
```

However, this still has the problem. To refresh everyone, the problem is 13.9.1(12), which says that that the call to Unchecked_Conversion is erroneous for scalar types if the result is invalid. Because of this, once you have made the call, it is too late to apply 'Valid.

Steve Michell claims that the intent is that this should work. Bob Duff differs, saying that this was intentional. The AARM supports Bob's position.

It is noted that a lot of existing Ada code assumes that this would work. If we adopt a solution of adding new generics or adding a parameter to Unchecked_Conversion, that existing code will remain erroneous. Moreover, vendors may take more advantage of that erroneousness, knowing that users can prevent that if they wish.

Tucker wonders if a special exception to the erroneous rule for ‘Valid would be better. People object that this requires tracking information all over the compiler to be aware of this exception.

Steve Baird comments that he prefers the new generics as they can be implemented purely in Ada.

Tucker claims that optimization of ‘Valid is wrong (echoing a comment of Robert Dewar in e-mail). An implementation should “assume the worst” when handling 'Valid. But how can we reflect that in the standard?

Tucker suggests a possible rule that eliminates the need to keep track of values throughout the compiler:

“For a scalar value produced by Unchecked_Conversion (or an imported subprogram, etc.) which is not a valid value of its target subtype, the use of such a value other than for assignment or ‘Valid is erroneous.”

Someone wonders about assignments that discard bits. Consider:

```
function UC is new Ada.Unchecked_Conversion (Integer, Natural);

type Rec is
    record
        C : Natural;
        B : Boolean;
    end record;
for Rec use
    record
        C at 0 range  0 .. 30;
        B at 0 range 31 .. 31;
    end record;

X : Natural;

R : Rec;

...

X := UC (-1); -- OK by proposed rule.
R.C := X; -- Discards the sign bit, OK by proposed rule;
          -- probably no check because the subtypes match.
```

After the last line:

- R.C’Valid is True
- X’Valid is False

Tucker claims that this is OK, and that other uses of both X and R.C are still erroneous.

It is noted that this would cause more code to be generated for 'Valid in some cases; Tucker notes that this is acceptable as long as nothing else is slowed down. The group agrees, but is not convinced that language wording can be created which expresses this.

Tucker will try to work on this wording.

Approve intent of AI: 4-1-1

AI-196: Assignment and tag-indeterminate calls with controlling results

First, we have a long discussion about when we discussed this previously. Eventually (with people searching old minutes), we determine that we never have discussed this.

The summary contains the complete answer to the problem.

The solution is to add another paragraph following 3.9.2(18) to include assignment statement. The summary should say this. There is no need to change 5.9(2), which works fine as it is.

Approve intent of AI: 6-0-1

Gary will make changes and add wording.

AI-228: Premature use of "must be overridden" subprograms

Pascal explains the current version of the AI. We could reintroduce the notion of these being abstract, but that was the way it was in draft 4 of Ada 95, and it was changed for draft 5. So he introduced the notion of "insubstantial" to handle this.

A long discussion about the meaning of abstract and "must be overridden" ensues. After several false starts, we look at the following example:

```
package P1 is
    type T1 is abstract tagged ...
    function F return T1;
    procedure P (A : T1) is abstract;
end P1;

with P1;
package P2 is
    type T2 is new P1.T1 ...
    -- Is P'access allowed here?? [Yes.]
private
    function F return T2;
    procedure P (A : T2);
end P2;

with P2;
package P3 is
    type T3 is new P2.T2 ... -- Legal (P is not abstract)
    function F return T3;
end P3;

with P1;
package P4 is
     type T2 is new abstract P1.T1 ...
     -- P'Access is not allowed.
end P4;
```

If the parent type has an abstract procedure, the inherited procedure is "must be overridden", but not abstract, so we can do anything with the inherited subprogram, and don't have to override it again when deriving from the derived type. The function case works the same, except that it is "must be overridden" when deriving from the derived type.

Bob Duff explains that "we" (the MRT) defined this so that if you "know a real definition is coming", you can take advantage of it (that is, the abstract restrictions do not apply to it). So it is intentional that you can use these as if they are normally defined, because they must be before the end of a scope. This is the reason that "must be overridden" items are not defined to be abstract.

Someone asks if P'Access freezes P? No, it could be in a default expression.

Pascal expresses concern about name resolution, but is convinced that a call from outside would need to use the real body, so this should work. Pascal will make sure that AI-211 does not conflict.

Tucker says that this AI is a tempest in teapot. He suggests that it should be a confirmation. Pascal says that we finally understand what these paragraphs mean.

Pascal will rewrite the AI.

Approve intent of the AI: 7-0-0

AI-242: Surprise behavior of Update

Discussed the changes to the AI.

Approve as written: 4-0-2

AI-246: View conversions between arrays of a by-reference type

Pascal notes that Rational solved the problem in the question by disallowing non-confirming component size clauses. Bob said that allowing such clauses is required by the recommended level of support.

Pascal is bothered by the view conversion restriction, because this is a chapter 13 problem. Why should we disallow conversions for a purely representation issue? It is noted that the restriction is only for structural conversion of unrelated arrays. Pascal withdraws his objection.

Are there more such problems? Steve B. and Pascal think that there are. Steve B. is given an action item to study the recommended level of support in chapter 13 to find and eliminate problems with aliased and by-reference types, creating an AI if necessary.

Several wording changes are suggested:

- In the wording, the fourth bullet should change "subcomponent" → "part" (part includes the object itself).
- Second bullet: "The target type and operand type shall have a common ancestor or neither type shall be limited."
- Fourth bullet: "In a view conversion, if the target type and the operand type do not have a common ancestor, then the component subtype shall not have a tagged, private, limited, or volatile subcomponent."

Tucker suggests moving the common ancestor wording to 4.6(9), and factoring it out of the following bullets.

After further discussion, Randy completes the wording during a break. Later in the meeting, the following wording is proposed to replace 4.9(9-12.1):

> "If the target type is an array type, then the operand type shall be an array type. The target type and operation type shall have a common ancestor, or:
>
> - The types shall have the same dimensionality; and
> - Corresponding index types shall be convertible; and
> - The component subtypes shall statically match; and
> - Neither the target type nor the operand type shall be limited; and
> - In a view conversion: the target type and the operand type shall both or neither have aliased components; and the operand type shall not have a tagged, private, limited, or volatile subcomponent."

Approve AI with editorial changes: 6-0-0

AI-247: Alignment of composite types

There is a discussion of the reasons for deleting the rule. This is a legality rule (meaning all users and implementations are bound by it); why should we prevent users from doing what they want, and implementations from supporting what they want? Moreover, there is no value to making this an implementation advice, because it is constraining only when you have explicitly specified something. The group agrees.

Approved AI: 5-0-2

AI-259: Can accesses to atomic objects be combined?

Implementation advice was proposed by Robert Dewar in the e-mail.

> "A load or store of an atomic object should, where possible, be implemented by a single load or store instruction which accesses exactly the bits of the object and no others. The implementation should document those instances in which it is not possible to follow this advice."

Straw vote on the intent of this advice: 4-0-4

Randy will create the AI.

AI-272: Pragma Atomic and slices

Pascal explains that his AI writeup simply says that a slice is not an atomic object. The only problem is for unconstrained array types, but that is not a real problem, as it would be very hard to support that.

An alternative would have been to say that an atomic object is one declared with the first subtype of an atomic type, but that would be incompatible.

Approve AI as written: 6-0-0

AI-279: Tag read by T'Class'Input

For case B, Tucker would prefer Program_Error. If it is an abstract type, there is no way that it could have been written to the stream, even by accident. Case A is something that could have been written (by writing an object of a type and reading it with the attribute of the wrong type).

The discussion turns to case C. Tucker notes that in this case, we've returned a tag representing a type that is in the class. So, why isn't this just an access to elaboration error (raising Program_Error)?

Steve Baird puts up an example:

```
procedure P is
    type T is tagged null record;
    package P1 is ...;
    package body P1 is separate; -- (1).
    package P2 is ...;
    package body P2 is separate; -- (2).
    ...
end P;

separate (P)
package body P2 is
    type T1 is new P.T with null record;
end P2;
```

```
separate (P)
package body P1 is
    ...
begin
    ... P.T'Class'Input ... -- (3)
end P1;
```

Assume that the call (3) reads a tag of T1. T1 has not yet elaborated. It hasn't even been created yet. That means that the state for determining whether T1 is elaborated (the elaboration bit) has not been initialized yet. So, you can't just raise Program_Error by checking that bit. To set it would require moving the elaboration bits (or at least their initialization) somewhere earlier (before (1), in this program).

That is very hard to do in the presence of separate compilation (as in the example). Since the bit is library-level, many implementations could do it by insuring a static initialization, but that doesn't fit well into the Ada elaboration model, and it doesn't help for nested types.

Randy points out that the problem with saying this is erroneous is that any use of a stream attribute before *all* of the descendants have been elaborated would potentially be erroneous. Since applications typically have no control over the contents of streams they read (files can be replaced, sockets can have garbage in them), that means that T'Class'Input cannot be used in library-level elaboration code.

The group feels that the call on Input should be erroneous before the type found in the stream has been frozen, because the distributed overhead of mandating a check is too expensive for a very rare case.

Approve intent of the AI: 6-0-0

Randy will rewrite the AI.

AI-280: Access to protected subprograms and entries after finalization

Steve Baird would like a similar rule for collections. He shows an example of why:

```
type T is new Ada.Finalization.Controlled with ...;
X : T;

procedure Finalize (X : in out T);
type Ref is access Task_Type;

procedure Finalize (X : in out T) is
begin
     ... new Task_Type;
end Finalize;
```

When this scope finalizes, Ref is finalized, then X. The finalization of X allocates in a finalized collection.

This is the same access after finalization problem.

Steve suggests raising Program_Error in this case; this requires a bit to determine if the finalization has happened for the collection. Randy and Tuck argue for making this case a bounded error, since we don’t want this overhead on all access types (such as access Integer). It's only a problem with access types that have controlled or task parts (using parts in the technical sense). Tuck suggests as an alternative just having it not finalize; this is rejected by the group as it would allow a program to violate the invariants of a controlled type.

So the following rule is suggested: Allocating from a collection after it is finalized is a bounded error. Either you get Program_Error or the objects get finalized normally. AARM remark: Special code is necessary only for library-level access types whose finalization has an effect. Finalize routines can only be defined at the library-level, and they are (currently, at least) the only known way to trigger this problem.

We turn to the wording for the protected case. The word “normally” should be deleted from the wording.

Steve Baird asks about deallocation during finalization of the collection. The same answer seems to work: set the "raise Program_Error" bit when you start, and raise Program_Error if it happens.

Steve Baird gets the AI for a rewrite. The group recommends to add plenty of examples to the AI.

AI-283: Truncation of stream files by Close and Reset

Randy describes the problem and the result of his compiler survey (most compilers use the same algorithm for truncation as for Sequential_IO: truncate on open for Out_File).

The current behavior of most compilers is a bug: it causes any data already in the file to be discarded, even if it is not going to be overwritten and even if the user has taken precautions to move the current index before closing. The group agrees that this is wrong.

Tucker comments that if you don't truncate on open, it is expensive to truncate at close on Unix systems. There are probably other systems with this property.

Pascal notes that if we don't truncate, we need to provide a way for the user to do it. Tucker notes that no existing implementation truncates on close, so we only need to provide what is currently available. Delete followed by Create would accomplish that.

Therefore, the group settles on no truncation of Stream_IO files, except of course by Create in In_File or Out_File modes. This should be accomplished by fixing the equivalence so it clearly doesn't imply truncation.

Approve intent of the AI: 3-0-3

Randy will write the wording.