Minutes of the 18th ARG Meeting

7-9 February 2003

Padua, Italy

Attendees: Steve Baird, John Barnes, Randy Brukardt, Alan Burns, Pascal Leroy, Steve Michell, Tucker Taft, Tullio Vardanega.

Meeting Summary

The meeting convened on 7 February 2003 at 9:15 hours and adjourned at 14:00 hours on 9 February 2003. The meeting was held in a conference room at the University of Padua, Department of Mathematics, in Padua (Padova), Italy. Tullio Vardanega hosted the meeting. The meeting covered the entire agenda. Friday and Saturday were primarily devoted to amendment AIs, while Sunday was primarily devoted to normal AIs.

AI Summary

The following AI was approved without changes:

AI-316/01 Return accessibility checks and value conversions (8-0-0)

The following AI was discussed and put on hold:

AI-284/04 Nonreserved keywords

The following AIs were approved with editorial changes:

AI-167/02 Scalar unchecked conversion is never erroneous (6-2-0)
AI-178/02 Which I/O operations are potentially blocking? (8-0-0)
AI-216/10 Unchecked unions -- variant records with no run-time discriminant (7-0-1)
AI-224/07 pragma Unsuppress (8-0-0)
AI-228/03 Premature use of 'shall be overridden' (8-0-0)
AI-256/06 Various wording changes to the standard (8-0-0)
AI-259/03 Can accesses to volatile be combined? (7-0-1)
AI-265/06 Partition Elaboration Policy (8-0-0)
AI-270/03 Stream item size control (7-0-1)
AI-283/01 Truncation of stream files by Close and Reset (5-0-3)
AI-287/02 Limited Aggregates Allowed (6-0-1)
AI-298/03 Non-preemptive dispatching (7-0-1)
AI-306/01 Class-wide extension aggregate expressions (7-0-1)
AI-310/02 Ignore abstract nondispatching subprograms for overloading resolution (6-0-2)

The intention for the following AIs was approved but they require a rewrite:

AI-204/03 Language interfacing support is optional (8-0-0)
AI-218-02/03 Accidental overloading when overriding (6-0-2)
AI-230/04 Generalized use of anonymous access types (6-0-2)
AI-279/02 Tag read by T'Class'Input (7-0-1)
AI-280/02 Allocation, deallocation, and the use of objects (7-0-1)
AI-285/02 Support for 16-bit and 32-bit characters (8-0-0)
AI-297/03 Timing Events (7-0-1)
AI-301/04 Missing operations in Ada.Strings.Unbounded (7-0-1)
AI-307/02 Execution-time clocks (8-0-0)
AI-311/01 Static matching of formal scalar subtypes (6-0-2)
AI-317/01 Partial parameter lists for formal packages (8-0-0)
AI-320/01 Violating Ada semantics with an interfacing pragma (8-0-0)
AI-321/02 Definition of dispatching policies (8-0-0)

The following AIs were discussed and require rewriting for further discussion or vote:

AI-217-05/01 Type stubs with abstracted packages
AI-217-06/01 Limited with clauses
AI-217-07/00 Child type stubs
AI-294/02 Instantiating with abstract operations
AI-296/01 Vector and matrix operations
AI-312/00 Environment-level visibility rules and generic children

The following AIs were voted No Action:

AI-261/01 Extending enumeration types (8-0-0)
AI-274/00 Requiring complete record representation (8-0-0)
AI-289/03 Truncation of real static expressions (7-0-0)
AI-308/01 Private generic children are private (8-0-0)
AI-319/01 Object_Size attribute (6-1-1)
AI-322/00 User-defined operator symbols (6-2-0)
AI-323/01 Allow in out parameters for functions (7-1-0)

Detailed Minutes

Meeting Minutes

We consider the minutes of the 17th ARG meeting. John Barnes asks for a change in the minutes: in the penultimate paragraph of AI-204, “working of” should be “wording of”. The minutes (with this change) were approved by acclamation.

Publicity of the Amendment

Pascal Leroy has submitted an article to the Ada User Journal. They are prepared to publish one such article per issues; we need more articles.

Alan Burns will write an article on the real-time features of the Amendment for the March issue. We’ll try to have an article on the OOP features of the Amendment for the July issue (victim to be chosen).

WG9 Action Item for Access Subtypes

We didn’t quite get the response we hoped for on this item. We now need to create a report for WG9. Pascal volunteers to investigate all AIs on this issue, and then work with Tucker to create a report.

Next Meeting

The next meeting will be held in conjunction with Ada-Europe and the WG9 meeting. That makes the dates June 20-22 in Toulouse France, with the meeting starting at 1 PM or after the end of the WG9 meeting, whichever is later.

We then discuss the following meeting. SIGAda is again too late (December), so we need a host for an October meeting. Randy and Steve Michell volunteer to host a meeting. Randy can’t give firm dates, as he would need to insure that we’re not meeting on a football weekend (lodging would be impossible on a football weekend). [For the record, October 3-5 is not a football weekend, while October 10-12 is]. Steve Michell’s offer of Sydney, Nova Scotia, Canada is chosen. After discussion, the dates of October 3-5, 2003 are chosen.

Motions

The ARG thanks our host, Tullio Vardanega, for hosting the meeting and treating us to one of the longest and most memorable meals in ARG history. Approved by acclamation.

Requests

Tucker requests that the action item list get released very soon, rather than with the minutes. This gets general approval.

Old Action Items

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

Steve Baird:

- AI-216
- AI-251

John Barnes:

- AI-204
- AI-218-02
- AI-296

Randy Brukardt:

- AI-178
- AI-218-01
- AI-224
- AI-259

Editorial changes only:

- AI-147
- AI-195
- AI-212
- AI-220
- AI-249
- AI-255
- AI-263
- AI-305

Alan Burns:

- AI-265
- AI-297
- AI-298

Bob Duff:

- AI-287

Pascal Leroy:

- AI-228
- AI-285
- AI-310
- Find a host for the February meeting and sent details to the ARG mailing list.
- Contact the Ada Europe organizers to include a presentation on the Amendment in the conference agenda.

- Write an article on the Amendment for the Ada User's Journal.

Tucker Taft:

- AI-230
- AI-317
- AI-324 [Physical units (length/dimensional analysis)]
- Give Amendment briefing and non-real-time workshop at SIGAda 2002.

New Action Items

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

Steve Baird:

- AI-51
- AI-109
- AI-291
- AI-294

John Barnes:

- AI-204
- AI-218-02
- AI-254
- AI-296
- Present AI-217 examples to Ada UK, report on comments.

Randy Brukardt:

- AI-217-05 (by March 15th, 2003)
- AI-279
- AI-280
- AI-292
- AI-301
- AI-320
- Create an extended example of the use of AI-217-05, etc. (by April 1st).
- Create a detailed design of the implementation of AI-217-05, AI-217-06, AI-217-07 in Janus/Ada (by May 15th).

Editorial changes only:

- AI-167
- AI-178
- AI-216
- AI-224
- AI-228
- AI-256
- AI-259
- AI-265
- AI-270

- AI-283
- AI-287
- AI-306
- AI-310
- AI-316

Alan Burns:

- AI-266-02
- AI-297
- AI-307
- AI-321
- Write an article for Ada User Journal on real-time features of the Amendment.

Gary Dismukes:

- AI-158

Bob Duff:

- AI-219
- AI-239
- AI-269
- AI-303
- AI-318
- Be the test creator of last resort.

Mike Kamrad:

- Various items to be standardized [jointly with Mike Yoder]
  - External_Tag
  - Storage_IO of tagged types
  - Array indexed by holey enumeration
  - Static elaboration
  - GNAT attributes and pragmas

Pascal Leroy:

- AI-217-06 (by March 15th, 2003)
- AI-264
- AI-285
- AI-311
- Create an extended example of the use of AI-217-06, etc. (by April 1st).
- Create a detailed design of the implementation of AI-217-05, AI-217-06, AI-217-07 in Apex (by May 15th).
- Write a report on access subtypes (with Tucker Taft).

Steve Michell:

- AI-148
- AI-250

Erhard Ploedereder:

- AI-237

Tucker Taft:

- AI-133
- AI-162
- AI-188
- AI-214
- AI-217-07 (by March 15th, 2003)
- AI-230
- AI-231
- AI-252
- AI-275
- AI-282
- AI-288 (split into two AIs: pre/postconditions and invariants).
- AI-290
- AI-293
- AI-295
- AI-304
- AI-312
- AI-317
- Create an extended example of the use of AI-217-07, etc. (by April 1st).
- Create a detailed design of the implementation of AI-217-05, AI-217-06, AI-217-07 in AdaMagic (by May 15th).
- Write a report on access subtypes (with Pascal Leroy).

Mike Yoder:

- AI-315
- Various items to be standardized [jointly with Mike Kamrad]
  - External_Tag
  - Storage_IO of tagged types
  - Array indexed by holey enumeration
  - Static elaboration
  - GNAT attributes and pragmas

Items on hold:

- AI-284 (waiting for more keywords to be defined)
- AI-298 (waiting for the completion of AI-321)

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 abstentions.

Detailed Review of Amendment AIs

AI-216/10: Unchecked unions -- variant records with no run-time discriminant

Steve Baird explains the changes to the AI. The major change is to use Bob Duff's rule, which essentially means "assume the worst" for equality. (The previous rule required a run-time check only for components that exist; that meant more run-time overhead for the check, and possibly more confusion for the user.)

A note was also added that Unchecked_Conversion is not the intended use of this feature.

Tucker does not like the wording of this note. The example is odd, no one would ever do this as written, which dilutes the power of the example. Change the example to `F1 : Float := 0.0;` and `F2 : Integer := 0;`.

Steve notes that he also added an Implement Permission for pragma Controlled.

Should say "that pragma Controlled be specified" instead of "that pragma Controlled is specified".

Approve AI with changes: 7-0-1.

AI-217-05/01: Type stubs with abstracted packages
AI-217-06/01: Limited with clauses
AI-217-07/00: Child type stubs

Where are we? The following alternatives and options have been considered. The number is the AI alternative number:

1 With type

2 Package abstract

3, 4, 5 Type stub; various options here:

- Where is the completion specified?
  - Never
  - Specify at the point of the completion [3]
  - Specify in the stub [4, 5]
- New kind of context clause to name the "incomplete" package? (Yes [5], No [3,4])
- Only one stub? (No [3,4, 5/01], Yes [5/02])
- Must be withed by completion? (No [3, 4, 5/01], Yes [5/02])

6. Limited with

7. type C.T ("child type stubs")

(Note: alternative [5/02] is an e-mail proposal to add a post-compilation check to alternative [5/01] which would require only one stub per completing type, and that the completing package **with** the stub's package.)

Tucker asks about issues common to all of these. This turns into a list of questions to be decided. The answers are summarized in the table below.

1. Does the completion know about incomplete type? (This is an implementation issue.)

Some implementations need to know whether there are any incomplete types around. That may change the choice of representations for access types, etc.

This should be yes; but an implementation can handle this with contortions or we could add an Implementation Permission. The group feels that this is important enough to some implementations that we have to require it.

2. Do incomplete and complete types have same fully expanded name?

   There is some implementation complexity (matching) if the incomplete and complete types may have different names. This is also somewhat related to question (9) below.

3. Do we have a new kind of compilation unit?

   - User level

     The cost here is a new kind of part for users to understand.

     Pascal repeats to the group a comment that Randy made at dinner the previous night: a user view of a package abstract would require users to decide where things go. It is not at all clear what goes in the abstract of a package and what goes into the visible part of a package. There is no obvious model to follow.

   - Implementation level

     The cost here is a new view for the implementation to understand and manage. For [1] and [7] there is probably a need (at least for some implementations) to manage a "ghost" package containing the incomplete type(s). For [6] there is a need to introduce two distinct views of compilation units.

4. Do we have a new kind of context clause?

   A new kind of context clause has some impact on "build" tools. It also adds some user complexity, as users must decide when to use the new context clause.

5. Can we avoid ripple effects relating to "indirect" with dependence?

   Adding or removing a distant with clause should not affect the local unit. Similarly, adding or removing a visible package rename should not affect the local unit (except for direct uses).

   The group agrees that this should be yes. So each proposal must include rules that prevent ripple effects.

6. Do we make this easy to use versus easy to implement versus easy to understand?

   A related question: Does the user of the completion know about any incomplete type?

   An attempt to answer the "easy to use" question turns out to be fruitless, as we need real users to answer that.

   We can discuss the implementation costs, based on gut feeling. Pascal and Steve Baird mention that in their case they would probably use the same technique for [6] and [7] (provided that the rules permit it) so these proposals would be equivalent for them in terms of complexity.

7. Is a child package inherent in the solution?

8. Are all dependencies visible
   - in context clauses?
   - or spec-body?
   - or parent-child?

     This is the SIGAda user's concern. We agree that the answer should be yes.

9. Can the solution handle the type "Object" naming style?

We agree that we must be able to handle that; not obvious whether a 'style' is good enough.

This issue is somewhat related to question (2) above.

10. Can the solution prevent access type proliferation?

    If you have declare your own access types in every point of use, then you need lots of (explicit) conversions.

    Type stubs [3,4,5] can avoid proliferation using an idiom similar to package abstracts. [7] can avoid proliferation by putting the access type in the parent. [1] and [6] don't seem to have a clean idiom. That's because the incomplete type cannot be exported along with the access type. Thus, it is necessary to **with** two packages in order to access the complete abstraction.

    Pascal disputes that. He says that [6] and [7] are not really different from that standpoint. In [7] you declare the access type with the incomplete type, in the parent. However with [6] you can effectively simulate [7] by having a **limited with** of one of your children. The parent then contains the access type, and you avoid proliferation in exactly the same manner. (Since [1] is very similar semantically to [7], it could also use this approach.)

    Tuck counters that while this is true, [6] is more complex than [7] and he doesn't see the need for this additional complexity. Randy points out that you still have to reference two different packages for these types, even if you avoid an extra context clause. ([7] also suffers from this.)

11. Works with nested packages

    Only [2] and [7] fully support nested packages. [6] supports it partially in the sense that you can reference type declared in nested packages. However, Tuck would like to support mutually-dependent types declared in packages nested within the same compilation unit. In [6] this could be supported if a unit were allowed to see a limited view of itself, but we agreed that a **limited with** of yourself is very strange methodologically and should be forbidden. Pascal wonders if Tuck's requirement regarding nested packages in a single compilation unit is really valuable.

    Although [3, 4, 5] in their current incarnation don't support a completion living in a nested package, Randy argues that it was an oversight when these proposals were written, and that this could be supported if needed.

The table below summarizes the answers to the above questions. Each column corresponds to a proposal, each line to a question.

	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	very hard	yes	very hard	very hard	5/01: very hard 5/02: yes	moderately hard	yes
2	yes	yes	no	no	no	yes	yes
3a	no	yes	no	no	no	no	no
3b	yes	yes	no	no	no	yes	yes
4	yes	yes	no	no	yes	yes	no
5	Answer not discussed and unknown by editor.	Answer not discussed and unknown by editor.	yes	yes	yes	Answer not discussed and unknown by editor.	Answer not discussed and unknown by editor.
6	medium	high (tools)	medium	medium	medium	high	medium

7	no	no	no	no	no	no	yes
8	yes (context clause)	yes (context clause, spec-body)	no	no	yes (context clause)	yes (context clause)	yes (parent-child)
9	yes	yes	no	no	no	yes	yes
10	yes (using ugly idiom)	yes	yes (using package abstract idiom)	yes (using package abstract idiom)	yes (using package abstract idiom)	yes (using ugly idiom)	yes
11	no	yes	possible?	possible?	possible?	yes (sort of)	yes

Tucker tries to explain why child packages could be inherent in this solution. Any time that you have mutually dependent types you have types that are somehow related. That means that they are all part of the same abstraction. A large abstraction should be mapped as a package hierarchy. Thus, the types all have a common ancestor. Even when that is not true, various renames will work.

Steve Michell says that it fits in well with package hierarchies; if the language had started out this way, it would be good. Tucker notes that the MRT had proposed something like this, but they didn't allow it in the visible part.

Pascal complains that this forces the users to use package hierarchies. Pascal also hates a parent knowing about its children. We specifically decided to avoid that in Ada 95; why should we reintroduce it here?

Steve Baird comments that they use a forward call to determine if the type uses finalization for incomplete deferred to body. He would like an implementation permission (at least) to allow requiring that the completing type exist. Others think that assuming the worst is not too bad here, and that such a permission is not necessary.

John Barnes asks if this meets the SIGAda concern. Pascal replies that this would be hidden deep within the parent specifications, it's not right at the top. So, no, it does not; it is very much like [4] in that sense.

Tucker explains the work-around to be able to do this at the library-level. (This is described in the Appendix of the AI.)

The discussion loses direction. We decide its time for a break.

After the break, we decide to try to narrow the field of contenders.

There is no opposition to killing with type [1], package abstracts [2]; type stubs without context clauses [3, 4]. Type stubs should be restricted to a single stub. This leaves single type stubs [5/02], limited withs [6] and child type stubs [7].

Pascal comments that type stubs will look like a wart five years from now. Randy rebuts that `type C.T;` seems inconsistent, as it's the only kind of type that will allow this notation. Tucker says that an incomplete type is not a type at all. That doesn't seem convincing; an incomplete type declaration certainly *looks* like a type declaration.

Consider the assignment `Q.X := P.all`; when the complete type isn't visible. When it is legal? All of the remaining proposals have this issue. Looking at the wording for AI-217-04, we agree that we need something like its rules for all proposals. Tucker says that the only difference would be that availability would be replaced by visibility for proposals [6] & [7]. Doing so would simplify the rules. Some members are skeptical, Tucker tries to convince them. (Later discussion suggests that visibility is not the right model.)

Tuck shows an example of how this issue relates to ripple effects:

```
package P is
    -- Q.T is incomplete (use the syntax of your favorite proposal).
    type Acc is access Q.T;
    X : Acc;
end P;

with Q;
package A is ... end A;

with P; with A;
package R is
    P.X.all; -- OK?
end R;
```

We all agree that the legality of `P.X.all` in `R` must not depend on the presence or absence of a `with Q` in `A`.

Tucker continues. Change A to read:

```
with Q;
package A is
    package MyQ renames Q;
end A;
```

In this case, `A.MyQ.T` is visible. If we're using a visibility rule, we don't want this to change the legality of `P.X.all`. (Score 1 point for type stubs!)

Can this problem occur with children? Yes, since you can access children through the renames. But, keep in mind that `A.MyQ.Child` is illegal unless Child is **with**ed, even if `A` had **with**ed `Q.Child`.

We now look specifically how this can happen for **limited with** [6].

```
package Q is
    type T is ...;
    Z : T;
end Q;

limited with Q;
package P is
    type Acc is access Q.T;
    X : Acc;
end P;

with Q;
package A is
    package MyQ renames Q; -- MyQ.T visible to units that with A.
end A;

limited with Q; -- (1)
with P; with A;
package R is
    P.X.all; -- OK? Must not depend on with of Q in A.
    A.MyQ.Z -- Is visible? (of course)
    Q.Z -- Is visible? (no, or we have ripple effects)
end R;
```

Consider the `limited with Q;` marked (1) in `R`. If it is absent, then evidently the name `Q.Z` is illegal. But if it is present, we don't want `Q.Z` to become legal just because we also happen to have visibility on the completion of `T` through `A.MyQ`. Otherwise we would have a maintenance problem where the addition/removal of `MyQ` would change the legality of `Q.Z`. So having visibility on the full view of Q is not sufficient to make `Q.Z` legal.

We try to figure out the wording, based on "hidden from all visibility" or "in scope" or "visible". "Hidden from all visibility" seems the most promising: there are places where the limited view is hidden from all visibility, notably at the places where there is a (normal) **with** clause for the unit.

Next, we look at an example with a library-level renaming:

```
with Q;
package QE2 renames Q;

with QE2;
limited with Q;
with P; with A;
package R is
    P.X.all; -- OK?
    QE2.Z -- Is visible(of course)
    Q.Z -- Is visible? (no.)
end R;
```

Does the presence of `QE2` change the legality of these constructs? Yes, you should be able to do the dereference. But you can't give `Q` extra visibility. So these rules require compilers to have two views of the package, and both have to be around.

Randy says that two views of the same package make the implementation much harder. Our implementation only expects a single view with a few well-defined places of visibility change. This is scattered all about. Pascal expresses agreement with Randy's assessment.

It seems better to make the **limited with** illegal in this case. That would eliminate the "two views" issue. But then we also have to make the **limited with** illegal if some **with** gives you direct visibility on a renames of the package. That doesn't seem too bad.

If you inherit a **limited with** in these cases (say from your parent or your specification), the **limited with** is "ineffective". Otherwise it is illegal. In both cases, all types are complete.

Steve Baird asks about a corner case. If a limited package is an instantiation, can you pass this as a formal package? No, because the **limited with** would be illegal. (A similar rule is necessary for C in the child type stubs proposal.) You shouldn't be able to tell that a **limited with**ed package is an instantiation.

For child type stubs [7], we do need a post-compilation check that the child exists. Then the look-ahead implementation of finalization mentioned by Steve Baird earlier would work. Someone mentions that Erhard didn't like the notion of requiring the completion to exist, but it is unclear what the rationale was. It is possible that this was related to a proposal that has been abandoned long ago.

Tucker suggests that we write these three alternatives up completely. We would also trade examples.

Tucker would like tagged incomplete to be extracted into a separate AI, so we don't have to consider it in each of these proposals. After discussion, we agree that that would be best.

Pascal would like to the implementers to do a detailed design on how they would implement each of the three proposals. That would put the implementation impacts in writing, which would cut hyperbole.

John asks if we could have the examples available for the Ada UK meeting in early April. We agree that the implementation designs should be finished by early May.

Randy will rewrite AI-217-05 as discussed.

Pascal will finish a write-up including wording for AI-217-06.

Tucker will create a write-up, including the missing visibility words for AI-217-07 (child type stubs).

All three will create an extended example, showing the use of their version of the AI (and possibly suggestions on the other versions).

All three will create detailed design documents describing how each of the three AIs would be implemented in their compilers.

The AI write-ups need to be finished by March 15th (so that the examples and implementation designs can be based on them). The examples need to be finished by April 1st (so that John can present them to Ada UK).

It would also be good to have input from ACT, unfortunately no one from ACT was present at the meeting.

AI-218-02/03: Accidental overloading when overriding

The problem that we needed to fix was overriding for private types. The different views can have different overridding characteristics.

John handled this by allowing ”redeclaration” of subprograms with “**maybe overrides**”.

John also added overriding indicators to renaming, since that was omitted in the previous draft.

Is it necessary to say something about the relationship of freezing to redeclarations? The group doesn’t understand the problem.

Randy writes the problematic example:

```
package P is
    type T is private;
    procedure Proc maybe overrides;
private
    type T is new Foo with ...;
    procedure Proc overrides;
end P;
```

Tucker objects to the “**maybe overrides**”. That’s just noise. Randy agrees.

Pascal worries that this is lying to the user. This surprises the group, as Pascal is Mr. Private. Clearly, the visible view does not know about any overriding, and allowing it to be visible would be breaking privateness. Later, Mr. Private says that he agrees that the visible part shouldn’t talk about overriding in the private part. But Pascal is still worrying about lying to the user.

We now try to answer John’s freezing question. Do you want the redeclaration to occur to before freezing? That seems like a good idea to be consistent with the language; also, compilers might want to do this checking at the freezing point.

Tucker comments that this is close to something he wants.

```
package P is
    type T is private;
    function “+” (L, R : T) return T;
private
    type T is new Integer;
    function “+” (L, R : T) return T renames <>;
end P;
```

The idea is to export the privately inherited operation, rather than re-implementing it. Tucker should submit a separate AI on this, as it doesn’t have anything to do with the purpose of this AI.

Tucker doesn’t like the way **overrides** reads. He would prefer using **overriding**. But it’s not clear where **abstract** goes. He writes all of the contexts of use for overriding specifiers. This turns into yet another syntax brainstorming session.

```
procedure P () is overriding abstract;
procedure P () is maybe overriding abstract;

procedure P () is [abstract] [maybe] overriding;
procedure P () is new overriding Gen ();
procedure P () renames Z overriding; -- Yuck
procedure P () is overriding renames Z;
```

We look at the prefix idea again:

```
overriding
procedure P () is abstract;

overriding
procedure P () is new Gen ();

overriding
procedure P () renames Z;
```

Looks OK. How does it work with **maybe**?

```
maybe overriding
procedure P () is abstract;

overriding
    optional
procedure P () is abstract;

possibly_overriding
procedure P () is abstract;
```

Another suggestion is made: having an overriding region, delimited either by new syntax or by pragmas:

```
overriding
    procedure P ();
    function F is new Gen ();
possibly
    procedure Foo ();
end overriding;

pragma Overridding (On);
    procedure P ();
    function F is new Gen ();
pragma Not_Overriding (On);
```

Pascal points out that both of these constrain the order of declarations, which probably are organized for the human reader. Randy comments that the **overridding** would get lost this way, because it will be pages away for the routines that it applies to. This is too important to lose that way.

Tucker doesn't like the syntax approach, because the checking is triggered by a pragma. Pascal replies that Tucker is always saying that we want this to look good, not bolted on. Randy says that this is repairing a problem in Ada 95, and people will turn it on and use it everywhere in new code. Pragmas will look bolted-on for this use.

Besides, if the syntax is given, it is always checked. The configuration pragma only changes the checking that occurs when the syntax is not used.

We decide to put the keyword in front. John is asked to find a clever idea for the optional case.

The check depends on what declarations the subprogram declaration can see. That depends on whether it was before or after the full view.

Steve Baird wonders why this applies to untagged types. It doesn't seem useful for them. But such a restriction would cause generic contract problems. Besides, Tucker thinks it might be useful.

Pascal would prefer that "redeclaration" be called something else, because it isn't declaring anything. Tucker suggests "overriding clause".

Should these require full conformance? That seems too strong. In practice, though, cut-and-paste is what will be used in the vast majority of cases, so the conformance doesn't really matter. Tucker argues that someone could look at the overriding clause and make a call on it; if we don't require full conformance, the parameter names and default expressions may not be correct. Leave it as full conformance. Pascal comments that this is more complicated than full conformance, though, as the overriding could be a generic instantiation, and the overriding clause should have a profile that corresponds to that of the instantiation. John will have to invent fancy new rules here.

Summary:

- Don't require maybe overriding on the partial view;
- Syntax at the front;
- Rules regarding redeclaration get defined, change name to overriding clause.

Approve intent of the AI: 6-0-2.

AI-224/07: pragma Unsuppress

The !corrigendum section is missing, it should be added.

Correct "the [the] point".

Correct the AARM Proof in 11.5(8): "An instance is {a} copy..."

Add the parenthesized note to the second last paragraph of 11.5(8): "omit the named check (or every check in the case of All_Checks)". Add the same text in the last paragraph.

Approve with changes: 8-0-0.

AI-230/04: Generalized use of anonymous access types

Tucker explains the changes to the AI. Primarily, he added wording.

The syntax for an object declaration is (where access_definition comes from AI-231):

```
object_declaration ::= access_definition [:= expression]
      | [aliased] [constant] subtype_indiciation [:= ...]
      | [aliased] [constant] array_type_definition [:= ...]
```

Constant was omitted because then "**access constant**" and "**constant access**" would mean different things, and Tucker hates that, it would be as bad as C++. Anonymous access objects are only allowed for constants, so saying **constant** is redundant.

The examples include AI-231 features; these should be removed. The AIs should be kept as independent as possible. access_definition should be defined here as well.

Fix spelling of "discriminant" in example.

Randy worries about the missing access variable (or the missing word **constant**); this syntax seems like a variable declaration. Having a constant without **constant** is surprising. He'd rather get rid of it if it isn't like existing variables and constants.

Why do we need a top-level object as opposed to just components? Because such an object can preserve accessibility, and a component cannot. The group agrees that this seems important for operating on access parameters. It is important not to lose the accessibility level of the parameter, because if it was lost, the run-time check may fail in some of the subprograms that you are calling.

Tucker doesn't think we'd want variables even if the syntax were better, because we wouldn't want access variables and access constants to have different accessibility. And this is necessary if we believe it is important to be able to preserve accessibility.

The words "variable object" was removed from 3.3.1(5); it seems necessary. Tucker replies that it isn't really necessary, since anything that is not a constant object is a variable object by 3.3(13). John thinks it has nice symmetry. Tucker points out that it isn't a legality rule; it should be defined in static semantics. The words probably should be added back there.

Returning to **constant**: Pascal says that he thinks that **constant access** isn't that confusing and that the distinction between **constant access** and **access constant** follows from ordinary English usage (unlike C++'s distinction between **T* const** and **const T***). Tucker disagrees, saying that adding the word will just show people that they can't have variables. We then repeat all of the arguments about that having variables. Randy wonders about boxing ourselves in here; if we adopt Tucker's syntax, we could never have variables, even if we later decided we needed them. We would have the same problem that we have with access parameters now (see AI-231).

Pascal says that for access parameters, the mode is **in** (which is always omitted)—which is a constant. But components are not constants, so the readability is a problem.

Pascal suggests that maybe we should use renaming for this.

```
X : access T renames A'Access;
```

Renames doesn't use **constant,** and that's not too good for readability (Pascal says that's a language screwup).

But perhaps *just* allowing renaming makes sense: it would be clearer that the accessibility level is preserved.

Tucker explains that he thinks that these would usually be used to save access parameters:

```
X : access T renames Param.D;
X : access T renames Param.X'Access
```

Tucker agrees that this is better, so does everyone else.

Approve the intent of the AI: 6-0-2.

AI-261/01: Extending enumeration types

Derived subprograms work oddly in this proposal, raising Constraint_Error for new values. That doesn't seem helpful. Nobody is enthusiastic about this capability anyway.

No action: 8-0-0.

AI-00265/04: Partition Elaboration policy

Alan doesn't remember the discussion about the prohibition of blocking items during task startup that is mentioned in the minutes of the last meeting. Yes, we had that discussion. This is just a deadlock situation, we don't need

special wording to cover it. This text should just be reduced to Implementation Advice (raising Program_Error in this case is pretty much like terminating the main subprogram).

Alan asks about discussion item (b) (dynamic attachments of handlers). Tucker suggests that if there are multiple dynamic attachments, the "last one" is executed. That seems to be what normally happens.

Tucker: We should allow implementation-defined elaboration policies. Just add ", or an implementation-defined policy".

Alan asks if we need to have priority inheritance when activating these. Yes, because otherwise a slow, low priority task could block the higher priority main task.

Approve intent of the AI: 6-0-2.

AI-265/05: Partition Elaboration Policy

Late on Saturday, Alan presents a new version of this AI with the changes requested.

In syntax, lower case "policy_". Also, change it when it is used in the wording.

Paragraph. 2 of Dynamic Semantics needs a comma after "fails". There should be no commas in the third sentence. "defered" => "deferred". The latter part of the last sentence should read "...then the most recent call on Attach_Handler determines which handler is attached."

The Legality Rule should be a Post-Compilation Rule.

Implementation Advice: "...recommended that {it} [that the active partition] be ..."

Syntax: "Concurrent is the default" doesn't belong here, kill it. (It has nothing to do with the pragma.)

Then add: "If there is no pragma Partition_Elaboration_Policy defined, or if..." with the rest of third paragraph of Dynamic Semantics, moved to the start of the Dynamic Semantics section.

Steve Baird thinks that we really need to say "Notwithstanding what this International Standard says elsewhere,...", because we're changing the standard semantics for various things.

Tucker would like to turn this around: "The execution of the declaration of Environment task is as defined in 10.2..."

We agree that this is best. The AI is returned to Alan for an update.

AI-265/06: Partition Elaboration Policy

Alan works all night on updating this AI, and comes up with yet another version on Sunday morning.

"...then the most recent call on Attach_Handler..."

Steve Baird wonders about "permanently blocked" in Implementation Advice. Change to "permanently blocked during elaboration".

Approve AI with changes: 8-0-0.

AI-270/03: Stream item size control

The wording says this is a representation attribute, the summary says it is an operational attribute. Which is it? It should be an operational attribute.

Steve Baird wants to know if the default value should be guaranteed to be a multiple of the stream element size. Yes. Change to "The value of this attribute is of type universal_integer and a multiple of Stream_Element'Size."

Steve Baird wonders why the first subtype is not used when determining when whether to raise Constraint_Error. Because you have to support T'Base; and the use of T'Base is sort of portable (there is a minimum guaranteed range).

John wonders why the summary says "only needs to write" and the wording says "read or write". Correct the summary.

Typos: in discussion: "advise" → "advises". "smaller" → "smallest" in the 13.13.2(17) quote. "if two implementation" → "if two implementations".

Approve AI with changes: 7-0-1.

AI-274/00: Requiring complete record representation

Pascal: Defining the effect of this pragma would be very difficult, because records are very complex in Ada—implicit components, dynamic sized components, and variants.

No Action: 8-0-0.

AI-284/04: Nonreserved keywords

Why did WG9 reject this? France (in particular, Jean-Pierre Rosen) was against it. Without the list of words, it was too hard to overcome the objection.

So we should leave this on the back burner, until we have a more definite set of keywords. Then we can resubmit it.

AI-285/02: Support for 16-bit and 32-bit characters

Pascal argues that Latin-9 is not widely used. He did a web search, and the only program he found using it was a version of EMACS. Most programs just directly support Unicode. So he believes that we should simply go to 16-bit source code.

The Unicode people have rules that we could follow. Most important are presentation characters, which should be filtered out. Also, they define "normalization" of characters. That merges characters that have the same meaning. That is separate from "case folding" (which converts between upper and lower case).

Note that string literals would have presentation characters filtered out, which is incompatible with Ada 95.

Tucker wonders why the Wide_Wide_Character literals are given as 00010000. These are just the character value between ticks, right? We don't want to have special cases for the weird things. This is described in 3.5.2(3) for Wide_Character; we need a similar paragraph for this. For 'Wide_Wide_Image, we really don't want to bulk up the runtime with these funny literals.

Steve Baird comments that the wording says "must have the same contents" as Wide_Character. What are the *contents* of Wide_Character? This should say "...has the same character literal or language-defined name as...".

Tucker would rather make presentation (other_format) characters illegal when they occur in a string or character literal. That seems more with the Ada philosophies, as it makes the incompatibility visible. That's essentially what is done for control characters in Ada. And it causes a compile-time error in the unlikely event that someone actually did this.

Approve intent of the AI: 8-0-0.

AI-287/02: Limited Aggregates Allowed

Steve Michell comments that words like “it’s a shame” and “unfortunately” in the problem statement are annoying. The problem shouldn’t appear biased toward a particular solution.

Might want to mention protected objects and private types as additional cases where the box is necessary. The “because it’s useful” is not very compelling, an example of why its useful for non-private types would be useful.

Pascal notes that the AI has the unfortunate effect that <> and (<>) have different semantics. Tucker writes an example on the board in unreadable yellow chalk:

```
type T is
    record
        X, Y : Integer;
        R : Rec := Blah;
        Z : Integer;
    end record;

(3, 4, <>, 7)   -- Use the default initial value for R.

(3, 4, (<>), 7) -- Which is the same as (3, 4, (others => <>), 7)
                -- Use the default initial values for components of R.
```

That suggests that (<>) is bad, because adding parentheses changes the semantics. So drop the (<>) shortcut.

Change the wording to be indented in the AI, so it will easier to read.

We discuss being able to modify limited constants with the Rosen pathology. (This is mentioned in the discussion section of the AI.) We decide it is not too bad. The only alternative is a distributed runtime check, which is clearly worse.

Approve AI with changes: 6-0-1.

AI-296/01: Vector and matrix operations

Tucker asks why the vector and matrix ranges are type Integer? Usually this is written as Positive, because otherwise you’ll get Integer’First in some places. Are negative indexes used? Probably, making it Positive seems unnecessarily restrictive, and it would be a change from the existing standard.

Wording: “involves an inner product” seems weird, but it will be necessary once the accuracy is defined with reference to the inner product.

John asks if we want to put this at the beginning of Annex G? No, renumbering all cross-references would be bad. Make this self-contained in G.3.

Do we need the non-generic equivalents? Yes, for all the reasons that we have those elsewhere, and they are no real cost to implementers.

John wonders whether to use “indexes” or “indices”. (Both are correct English usage.) Randy and Tuck search the RM. Indexes is not used, indices is used in various places (most importantly, 3.6(12)). So use indices.

Accuracy issues. Pascal says that various complex operations have a box error. These operations have cancellation. Use something similar for inner product if possible. John will talk to Terry Froggatt to get the perspective of a numerical analyst.

Pascal wonders if other operations should be included. John says no, because the error analysis starts getting ugly. But it is odd to not have Inverse in a matrix library. Pascal argues that we could have very loose accuracy requirements for fancy operations.

What operations would be useful?

- Inverse
- Eigenvalues, eigenvectors
- Determinant

Beyond that, it seems like overkill. John will look into these operations.

AI-297/03: Timing Events

Alan says that the main open issue is the AI-303 issue.

Pascal notes that Randy had suggested defining the semantics based on finalization, and using a restriction to handle the library level requirement for safety critical systems. That eliminates the entire semantic issue. Alan agrees that that would work. There is no objection to this idea.

The restriction should be No_Local_Timing_Events to be similar to protected objects.

"As the final step of finalization of a timer object consists of the ..." But it shouldn't call Cancel_Handler, just do those actions.

The definition of "potentially suspending operation" should be under dynamic semantics. That's easy, just move the header.

The wording for D.15(1) should say "{protected} procedures to be executed...".

The bounded error needs a description of the possibilities. If there aren't any other than Program_Error, then it should be described as a check.

Typos: "it return{s} true". Also, "False", not "false".

"An exception propagated from a handler has no effect", not "is ignored".

Tucker asks if it is "canceled" or "cancelled". Randy reports that the RM uses "cancelled" (after a search of the RM text).

Pascal notes that there doesn't seem to be any description of what these routines do in the normal case; the text describes what they do when they fail. It is noted that the proposal is clearer describing the routines than the wording.

More wording improvements:

"when the timing event is set" → "after the timing event is set".

"If several timing events are registered for the same time, they are executed in FIFO order."

The parameter names should be words; Interrupts use "New_Handler", "Old_Handler". Use "Event" instead of "TE".

Approve intent of the AI: 8-0-0.

AI-298/03: Non-preemptive dispatching

As discussed previously (for AI-321, which is now at the end of these minutes), this would now be D.2.4.

Change “Attack_Handler” to “Attach_Handler”.

Steve Baird wonders if a delay really needs to be a task dispatching point. Yes, that’s the whole point. The user can control dispatching by appropriate placement of `delay 0.0;`.

Approve AI with changes: 7-0-1.

AI-301/04: Missing operations in Ada.Strings.Unbounded

Randy discusses the state of the AI. There have been concerns raised (via e-mail) about any changes that are incompatible here. The added versions of routines with Unbounded_String parameters could cause problems in obscure cases. He also points out that according to Robert Eachus, the original intent of the package was purely for a convenient storage of strings; operations were intended to be done using type String. Thus, the primary problem is the fact that it is quite inconvenient to convert values in and out of the type. To_Unbounded_String is very wordy, and obscures the real purpose of the code, particularly in use-adverse environments.

The group agrees that the added parameter for Index is valuable; the alternatives are kludgy at best, and can be very expensive on very large strings.

Tucker suggests to changing the names of new routines to avoid having any new overloads with the same number of parameters. (Or remove the new routines if they aren’t important.) Make sure that the names selected are different for all three string packages (some users use multiple string packages).

Also, add a “+” for to simplify conversions.

Approve intent of the AI: 7-0-1.

AI-307/02: Execution-time clocks

The parameter names are short, that’s not the usual Ada practice. It is pointed out that they are the same as those in Ada.Real_Time, and this is intended to be similar. At least one member says that Ada.Real_Time uses those names only because of insufficient review, and that those are a bug. But its obviously too late to change them now.

Change the wording: “When a timer is destroyed...” → “When a timer is finalized...”

Tucker gives the standard lecture on the use of “shall”: “shall” is not used for implementation requirements. “Shall” is a requirement on the programmer, not on the implementer. So some of these should be written as “is” or “are”.

The last sentence of the first paragraph after the specification is a documentation requirement, and should be described as such. An alternative, which is preferable to a documentation requirement, is to say “It is implementation-defined which task, if any...”, as “implementation-defined” implies a documentation requirement.

Alan notes that the execution time value is set to zero at some unspecified point between the task creation and the end of the task’s activation. Someone asks why you would want to wait to the **begin**; at this point the user elaboration code has already executed, and it could be lengthy. So this should say the “beginning of the task’s activation”. This is especially important, as users may want to read the CPU clock as part of the elaboration.

Finalization here implies that many implementations will need a controlled resource. Yes, that’s true, but *controlled resource* doesn’t necessarily mean a controlled type. After all, protected objects are controlled resources (as they have non-trivial finalization). So finalization-adverse implementations can avoid using “real” controlled types, just as they do for protected objects.

Someone asks about the paragraph:

> "When a Timer is created, the resources required to operate a CPU-time timer based on the associated execution-time clock shall be allocated and initialized. If this operation would exceed the limit of the maximum number of timers in the system, the Storage_Error exception is raised. The timer is initialized in the disarmed state."

This is way too specific. It says when the initialization is done, which isn't necessary. And Pascal thinks that Storage_Error is wrong here. This case seems more similar to files (which raise Use_Error if there are not enough available). Do we need to say *anything* about this here? Can anyone handle this usefully?

The conclusion is that the first two sentences are relatively useless, perhaps they should be deleted. Alternatively, create a new exception. But then we should have dedicated exceptions in lots of other packages (like Timer). But if this is felt to be important, then add a specific exception for this case.

We take an inconclusive straw poll: Removing the paragraph gets 3 votes, leaving the paragraph gets 4 votes, and one person abstains.

Finalizing a Timer implies Disarm. That should be said.

Alan says we probably want a restriction No_Local_Execution_Timers. No, this is not necessary, because Timer a protected type, and that is covered by the existing No_Local_Protected_Objects.

Alan says that one reviewer doesn't like the names, because they imply errors when the timer expires while that may be part of the normal operation. We consider the names of the entities.

The exception should be Timer_Error, because we have a Time_Error. The reasons why this exception is raised are really errors, so the name should include _Error.

"Exceeded" does have a negative connotation, so the proposed change is an improvement. Use Timer_Expired instead of Time_Exceeded and Timer_Has_Expired instead of Time_Was_Exceeded.

Tucker isn't sure that is right. He draws a state diagram. There are three states (Disarmed, Armed&Expired, Armed&Not Expired). These seem consistent with the names.

Steve Michell asks if we want an access type here. No, because determining the collection and the accessibility level needed are problems. It's better to let the users declare their own.

Approve intent of the AI: 8-0-0.

AI-310/02: Ignore abstract nondispatching subprograms for overloading resolution

Pascal explains that this can be implemented this as a filter before resolution. That is, remove all abstract non-dispatching routines before resolution.

Steve Baird wonders why this isn't more general. It seems warty. Much discussion ensues.

Randy worries that it might be hard to find the right place for filtering. Tucker and Pascal try to convince him. Randy decides that he can't tell for sure without inspecting the compiler code.

Pascal says that there isn't a problem for generics, because there isn't a problem for private types. If it is an untagged view, then this applies; otherwise it doesn't. The same rule applies for generics.

John says that the only alternative is to define a whole new concept. He suggests:

```
function "*" (Left, Right : Glarch) is not;
```

This doesn't meet with too much enthusiasm.

Tucker suggests changing the wording, leaving the original text alone. And then add: "The name or prefix shall not resolve to denote an abstract nondispatching subprogram."

Approve AI with changes: 6-0-2.

AI-317/01: Partial parameter lists for formal packages

John looks at the example and wonders if we should get rid of the ugly **is end** from signature packages. This causes some discussion about empty packages, and eventually we conclude it just isn't worth fixing.

Steve Baird asks about the Formal.Formal.xxx problem. Tucker was directed to fix that at the Bedford meeting. He would like to do that in a separate AI. That's OK, as long as he does it.

We look at the issue of which names are visible outside an instance:

```
generic
    type T is new A;
      -- function F1 (X : T) return T; -- Implicit
    type U is new B;
      -- function F2 (X : U) return U; -- Implicit
package GP is ...

with package FP is new GP (T1, others => <>);
      -- Can use FP.U, can't use FP.T.
```

So FP.F2 is OK, and FP.F1 is illegal. This needs to be made explicit in the AI. Thus the 12.7(10) wording needs to include copies of inherited operations. Fix the discussion, especially eliminate "can we assume the implementers do the right thing".

John says that there is a semicolon missing in the example.

Tucker wonders if the wording fixes the Formal.Formal.xxx problem. No, but the change would go here. Humm, better not change that separately; it does need to be included in this AI.

Approve intent of the AI: 8-0-0.

AI-319/01: Object_Size attribute

The problem section is missing from the AI. Randy attempts to explain. For scalar types, the "value" size and the "object" size are often different. It's necessary to be able to specify these with more control than is available for 'Size. And GNAT has it.

Pascal argues that there is more than one "object" size. Components, top-level objects, aliased top-level objects don't need to have the same size.

Probably need to just say aliased object of the subtype by default. When specified, it changes all aliased things, and *should* change other components.

Pascal and Steve Baird are adamantly opposed to allowing specification of this attribute for non-first subtypes. The reason is that this would cause trouble with static matching for general access types. Subtypes with different representations would not match, and Pascal argues that this would break the principle that the high-level semantics of the language should be independent of the representation (this principle is not adhered to everywhere, but he doesn't want to add more of this). It is pointed out that the AI does not *require* that you support specifying this attribute for non-first subtypes. Pascal and Steve Baird claim that even giving just a permission to do this would change the model of the language, and that in this case it's better not to standardize the attribute, and leave it to implementations to do what they want.

Steve Baird wonders about specifying size for abstract types, and thinks those should be excluded here. No, that's useful for roots of tagged trees, as abstract types may have concrete components.

Pascal objects to specifying it covering more items, as then you can't give a confirming clause. Tucker claims that that is already true for 'Size, it is not new for 'Object_Size.

We vote the AI No Action: 6-1-1.

AI-322/00: User-defined operator symbols

Pascal notes that there are two main questions: How do you define these lexically? Is this a good idea methodologically?

Tucker thinks that more control over predefined operations (indexing, numeric literals, etc.) would be more valuable than adding a few additional operator symbols. Pascal thinks that's way over the top.

Steve Michell comments that he's had programs where he'd like implication. He suggests allowing symbols from the Unicode set only.

Pascal says that there are places where not having user operator symbols impacts readability.

```
MySetOfFoo.Union (Var, MySetOfFoo.Intersection (Var1, Var2));
```

Tucker doesn't buy it. He doesn't think that these sorts of uses are going to be common enough for the readability gain to be valuable. Most of the interesting operations are not unary or binary.

It appears that there are as many ideas on what would be valuable here as there are ARG members. There appear to be three main camps: very limited new operators from the Latin-1 set, new operators from Unicode characters only, operators for existing predefined operations. There doesn't appear to be anything resembling a consensus.

We vote the AI No Action: 6-2-0.

AI-323/01: Allow in out parameters for functions

Randy explains some of his reasons for wanting this. (These are described in the AI.)

Steve Baird points out that the wording of 6.6(3) isn't quite correct. The instantiation case needs to specify that there are no other parameters.

Several people remark that this is too late.

We vote the AI No Action: 7-1-0.

Detailed Review of Regular AIs

AI-167/02: Scalar unchecked conversion is never erroneous

Tucker reads the new rule.

Steve Baird writes an example why he believes that this rule is a problem.

```
type Small is range 0..255;
for Small'Size use 8;
function Cvt is new Unchecked_Conversion (Integer, Small);
type A array (...) of Small;
X : A;
```

```
X(I) := Cvt(12345);
if not X(I)'Valid then ...
```

Steve Baird is concerned that assignment might truncate the value.

Tucker claims that the requirement that the sizes match for Unchecked_Conversion avoids problems of truncation. For instance, the example given in the minutes of meeting 15 is already implementation-defined.

Steve Baird asks if

```
X := U_C(Y);
if X'Valid then
    Assert (X = U_C(Y));
end if;
```

always has to be true. Tucker thinks so if the sizes match.

Steve Baird would like to be able to use Unchecked_Conversion to give more information to the optimizer. That seems wrong, that's what 'Valid is for.

Pascal wonders if this really solves the user problem. What about writing a constant? If we say "assignment statement", we do not cover initialization of variables and constants, and that's surprising to the user. If we say "assignment operation", that covers a lot of ground. Are function results and parameters passing OK? Tucker says that he didn't intend that, but that the wording does indeed have that ramification.

So the wording should say "assignment operation directly as part of an assignment statement or stand-alone object initialization." It does not cover subcomponents or function calls.

That's awfully wordy. Tucker thinks that writing it in terms of syntax would be better. "It is the expression of an assignment statement or an object declaration."

Steve Baird says that he still hates this, because Rational's implementation approach is to try to create the invariant that objects are always have a valid value. But the AI makes such an implementation incorrect. Several members claim that such an approach is already incorrect. Steve convinces them that it is possible, but not that it is friendly or desirable.

John makes a very strong statement to the effect that the UK really wants this problem solved.

Someone wonders if this covers imported functions? Yes, this wording also covers that case. Pascal wonders if it should cover other things as well, like machine code. But in those cases, you don't have to do a read first, so you can apply 'Valid before your program goes erroneous.

The paragraph number that needs to be changed is 13.9.1(12); this is wrong in the AI.

The AI is in the form of a confirmation, it needs to be changed to the form of a binding interpretation. The title also should be changed.

Approve AI with changes: 6-2-0.

AI-178/02: Which I/O operations are potentially blocking?

Typo in the 3rd paragraph of the response: "narrow interpretation".

Steve Baird wonders if we are going to change the wording as per the editor's note. No, it's insufficiently wrong.

In the 7th paragraph of the response, change "that would wrong" → "that would be wrong".

Approve with changes: 8-0-0.

AI-204/03: Language interfacing support is optional

John explains the wording changes he made.

Pascal wonders about Rational's Fortran implementation, which doesn't have Logical. He claims that Rational's customers needing the Fortran interface do not need Logical. He would prefer to use the wording as for specialized-needs annexes from 1.1.3(17). After discussion, the group agrees that that is preferable. Rational's customers are better off with a compile-time check by omitting Logical than a runtime problem caused by an incorrect implementation.

John gets the AI back for another update.

Approve intent of the AI: 8-0-0.

AI-228/03: Premature use of 'shall be overridden'

Only wording change is to say that renaming a 'must be overridden' is illegal.

Tucker asks that the wording be changed from "is a subprogram" to "denotes a subprogram".

Approve AI with changes: 8-0-0.

AI-256/06: Various wording changes to the standard

We discuss all of the wording changes in detail.

Change (7) should also cover D.3(15), D.2.2(17), which have the same lousy wording.

Corrigendum section should say D.4(15), not D.4(5).

Approve AI with changes: 8-0-0.

AI-259/03: Can accesses to volatile be combined?

There are objections to the normative wording change. After discussion, we decide to kill the normative wording, because it doesn't add anything. And it would be hard to do on a caching CPU. The IA requires documentation of any cases where anything other than the object is accessed, and that is better anyway.

Steve Baird notes that the IA needs to mention alignment as well, as the wording as it is covers an unaligned 8-bit access. That clearly isn't intended.

Pascal wonders about the "old" wording of "writing directly to memory" (C.6(16)). That seems to be hard to guarantee if you have caching. Tucker says that if a cache is consistent, then it can be considered to be the memory. Pascal is not convinced. We eventually agree to not answer a question not asked.

Approve AI with changes: 7-0-1.

AI-279/02: Tag read by T'Class'Input

In the corrigendum wording for 13.13.2(34), correct "identifed" to "identified".

We discuss the "never been frozen" wording (3.9(12)). Randy explains the problem; we want this routine to never be able to return a tag for a library level type that is not frozen. Tucker thinks that it is better to specify it for library-level types only. For other types, we should just make it unspecified.

This happens at the freezing point, because we don't want to make an object of the type before it is frozen. That is a point in the elaboration, which is a dynamic point. It would be best to say that somehow, as freezing is a compile-time notion.

The new Erroneous Execution wording should be changed to: "If the internal tag provided to T'Class'Input identifies a specific type that has not been frozen, or does not exist in the partition at the time of the call, execution is erroneous."

Approve intent of the AI: 6-0-2.

AI-280/02: Allocation, deallocation, and the use of objects

Steve Baird describes the issue and the solution. He mentions that it is important to be careful to note the difference between "start of finalization" and "end of finalization".

Looking at the example in the discussion, "Ref's" should be "T2_Ref's".

Tucker doesn't understand why this would be a problem; do we really care about library-level finalization? The memory of the program will be recovered by the OS (or a restart) in any case. We do care, because resources other than memory may need to be recovered. OSes don't always recover resources properly. Can this really happen? Yes, the original problem came from Claw via GNAT, so it obviously can happen in a real system.

In the summary, correct "bound error" to "bounded error". Remove the parenthetical remark, it's misleading, as any such ignored finalization can have no effect.

Similarly, in the wording for 4.8(11), remove the part after the semicolon in the "Bounded Error" paragraph.

Tucker would prefer to require the check only for allocated objects with non-trivial finalization, not for class-wide types. Since the check is part of the allocator, that is always known. He would prefer to be required to make the check in as few cases as possible. The reason for including class-wide types is that some extension might include a controlled or protected part. But that's not a problem if we talk about the allocated object as opposed to the type specified. "If the object created by the allocator has a controlled or protected part".

Pascal comments that the corrigendum change seems to replace an unchanged paragraph. It would be better to avoid that as it could confuse the readers of the RM.

Approve intent of the AI: 7-0-1.

AI-283/01: Truncation of stream files by Close and Reset

The wording refers to A.8.2(28), which should be A.12.2(28).

For End_of_File, the equivalence is still wrong, because we can change the position. The wording in A.8.3(9) does take that into account. Better to define End_of_File directly, modeled on Direct_IO.End_of_File (A.8.5(15)). Equivalences never work anyway.

John: the !qualifer is "clarification", which seems wrong. It should be "error".

This isn't compatible with some implementations, but the fix is easy for a program that expects incorrect behavior.

Approve AI with changes: 5-0-3.

AI-289/03: Truncation of real static expressions

The short description of the problem is that 'Machine_Rounds when False does not specify much, because it is a combination of various failures. Thus, specifying what happens in that case can provide the wrong answer.

Pascal doesn't understand the problem. If you have:

```
X : constant T := 1.0/3.0;
```

That's the same from the compiler's point of view as

```
X : constant T := 3#0.1#;
```

It's just a value. So what does it mean to do what the machine does? We don't know (in general) the sequence of operations that got us this static value, and we don't want the result to depend on that anyway. Any change in this area would be dramatically incompatible, as there are many programs out there that depend on exact evaluation of static expressions.

Another way to look at it is to consider the machine numbers M1 and M2 nearest to 1/3.

```
---M1----1/3--M2---
```

The machine can't represent 1/3, the value will be either M1 or M2. So how can we usefully say what the machine would have done in this case?

The conclusion is that the question is bogus. Pascal's argument is convincing.

We vote the AI No Action: 7-0-0.

AI-294/02: Instantiating with abstract operations

This AI grew out of an e-mail discussion between Tucker and Pascal.

We agree that the instantiation of (P2.T2) ought to be illegal, and perhaps that should be marked in the question's example. Pascal continues to claim that the RM wording does not make this illegal. Others disagree, saying the "corresponding" is not a technical term in this use.

Pascal explains the question's example (see the AI):

> T1 abstract; P(1) concrete.
>
> Formal D derived from T1; P(2) concrete.
>
> T2 abstract derived from T1; P(3) concrete; P(4) abstract, overrides P(3).
>
> P(3) corresponding to P(2) and P(1), not P(4)

Tucker says that P(3) is gone, you can't "correspond" to it. Steve and Pascal say no, it's still around, you just can't name it. Tucker does not agree with this analysis.

Tucker now suggests that if the wording can be improved with simple changes, that's would be fine. The group agrees.

Steve Baird will try to fix the words to eliminate this confusion.

AI-306/01: Class-wide extension aggregate expressions

There are two solutions in this AI. The first solution is a legality rule, the second one changes name resolution. We don't want to change name resolution; the check should be a legality rule. Besides, the second solution doesn't solve the problem, as indicated in the AI ("it's not clear that the second proposed solution will work").

So, we're adopting the first solution.

Also, kill off most of the discussion about the second solution.

Approve AI with changes: 7-0-1.

AI-308/01: Private generic children are private

The original proposal asks for private instantiations. Tucker points out that any such instantiations would be invisible, what would they be good for?

We vote the AI No Action: 8-0-0.

AI-311/01: Static matching of formal scalar subtypes

We agree that the "never match" semantics is preferable here. However, there is no wording.

Pascal will write wording and update the AI.

Approve intent of the AI: 6-0-2.

AI-312/00: Environment-level visibility rules and generic children

This clearly should be legal.

Tucker will write the wording to make it legal.

AI-316/01: Return accessibility checks and value conversions

Approve AI: 8-0-0.

AI-320/01 Violating Ada semantics with an interfacing pragma

Wording: should start "Conventions..." instead of "A convention..."

The problem can occur for *any* import or export, not just foreign ones. (If an assembler routine uses the Ada calling convention, that makes no guarantee that it doesn't violate Ada semantics.) Can we just say any interfacing pragma? Randy says no, because we have to allow pragma Convention (Ada) not to break semantics. Otherwise, any explicit confirmation of the convention would allow the compiler to do anything at all. Why not exclude Convention at all? Because a foreign convention may break Ada semantics. Consider a C calling convention that can't propagate an exception; we certainly want to allow those and have them covered by this paragraph.

Export and Import can cause trouble for any convention, Convention can only cause trouble for foreign conventions.

Tucker doesn't like the wording. He suggests saying "defined in subclause 6.3.1". No, that defines all conventions, and wouldn't be very useful. Perhaps, it would be better to list all of the Ada ones.

Approve intent of the AI: 8-0-0.

AI-321/02 Definition of dispatching policies

This AI rewrites D.2.1 to move preemption to D.2.2. That's necessary so that a non-preemptive policy can be defined. Alan explains that Ted rewrote his original proposal.

This should be an ordinary AI; we're just fixing the over-specified description of dispatching points.

Steve Baird comments that the new D.2.1(8) doesn't make sense: "...a (possibly more than one)...". Tucker suggests that this sentence belongs in D.2.1(6), because you should say that it goes back to a queue before you talk about choosing it. "When a task reaches a task dispatching point, it goes back to one or more ready queues. And then one task (possibly the same task) is selected..." Paragraph 8 is then deleted.

D.2.1(7) should say "...on the appropriate ready queue{s}...", because more than one is possible.

D.2.2(3): The policy_identifier is either a language-defined policy identifier or an implementation-defined identifier.

We discuss where new policies should go. We don't want these to look bolted-on, put new policies in D.2.3, D.2.4, etc. Pascal suggests moving FIFO_Within_Priorities to D.2.3. Alan objects, as that would change the section of the policy for no important reason. The group agrees with Alan.

Alan explains that D.2.2(13) covers everything necessary for the existing FIFO_within_Priorities policy. After discussion, the group agrees.

The title of D.2.2 should remain the same.

John asks again that D.2.2 be split. It's too confusing to mix the general information about specifying priorities and a specific policy. Alan again expresses concern about renumbering. Randy reports that the only cross-reference to D.2.2's specific policy is in D.2(1), and that paragraph is wrong anyway if there are additional policies. So the group changes its mind and says that this should be split, with appropriate title changes.

Tucker still is confused about D.2.1(7) (last sentence), D.2.2(13) and the new D.2.2(13.1)

He writes the three sentences on the chalkboard.

> When a higher-priority task is dispatched to the processor, and the previously running task is placed on the appropriate ready queue, the latter task is said to be *preempted*.
>
> In addition, when a task is preempted, it is added at the head of the ready queue for its active priority.
>
> A task dispatching point occurs for the currently running task of a processor whenever there is a non-empty ready queue for that processor with a higher priority than the priority of the running task.

He thinks this is backwards. He thinks that these three sentences should be combined into one well-crafted paragraph. The group agrees in general, Alan will attempt to do this.

The title of D.2.2 probably ought to be "pragma Task_Dispatching_Policy".

Approve intent of the AI: 8-0-0.