# Minutes of the 22nd ARG Meeting

5-7 March 2004

Phoenix, Arizona, USA

**Attendees**: Steve Baird, John Barnes, Randy Brukardt, Gary Dismukes, Kiyoshi Ishihata, Pascal Leroy, Erhard Ploedereder (except Sunday afternoon), Ed Schonberg (except Friday morning and Sunday afternoon), Tucker Taft, and Joyce Tokar.

**Observers**: Matthew Heaney (Saturday).

## Meeting Summary

The meeting convened on 5 March 2004 at 09:00 hours and adjourned at 15:00 hours on 7 March 2004. The meeting was held in conference room B at the Courtyard by Marriott Airport Hotel, in Phoenix, Arizona. Joyce Tokar of Pyrrhus Software hosted the meeting. The meeting looked at a long list of dormant AIs, as well as covering most of the amendment AIs on the agenda, as well as a few normal AIs.

### AI Summary

The following AIs were approved:

AI-100/03 Truncation required for static expressions if Machine_Rounds is False (9-0-0) [returned to a work item after the meeting.]
AI-200/01 Generic formal subprograms as dispatching operations (9-0-0)

The following AIs were approved with editorial changes:

AI-148/01 Requeue of protected entry calls (5-0-4)
AI-209/02 pragma Reviewable; can objects become uninitialized? (9-0-0)
AI-245/01 Consistency of inlined calls (8-0-1)
AI-251/11 Abstract interfaces to provide multiple inheritance (7-0-1)
AI-286/05 Assert pragma (7-0-1)
AI-329/03 pragma No_Return (9-0-1)
AI-351/03 Time Operations (9-0-0)
AI-360/03 Types that need finalization (8-0-0)

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

AI-291/02 By-reference types and the recommended level of support for representation clauses (5-0-2)
AI-302-03/02 Container library (8-0-2)
AI-344/01 Allow nested type extensions (8-0-2)
AI-345/04 Protected and task interfaces (7-2-1)
AI-362/02 Some predefined packages should be recategorized (9-0-1)
AI-363/02 Eliminate access subtype problems (9-0-0)
AI-364/01 Fixed-point multiply/divide (8-1-1)
AI-366/01 More liberal rule for Pure units (10-0-0)
AI-368/01 Restrictions for obsolescent features (8-0-2)

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

AI-318/03 Returning (limited) objects without copying
AI-359/02 Deferring freezing of a generic instantiation
AI-370/01 Add standard interface for environment variables
AI-373/00 Undefined discriminants caused by loose order of init requirements

The following AIs were voted No Action:

AI-58/08 Accessibility rules for Shared_Passive Packages (9-0-0)
AI-172/04 Questions about main subprogram (9-0-0)
AI-187/00 Task attribute operations are atomic but not sequential (9-0-0)
AI-191/02 Are formal parameters passed by reference objects or views? (9-0-0)
AI-218-01/06 Accidental Overloading while Overriding (10-0-0)
AI-218-02/04 Accidental Overloading while Overriding (10-0-0)
AI-222/01 Pragma Feature (10-0-0)
AI-232/01 Dispatching operation visibility and ambiguity (10-0-0)
AI-244/01 Legal units which depend on illegal units (9-0-0)
AI-250/03 Protected types, Extensible, Tagged, Abstract (10-0-0)
AI-266-01/05 Task Termination procedure (10-0-0)
AI-288/02 Pre- and Postconditions (8-0-0)
AI-290/01 Declaring functions Pure (9-0-1)
AI-292/00 Sockets operations (10-0-0)
AI-302-01/07 Data structure components for Ada (10-0-0)
AI-302-02/02 Container library (10-0-0)
AI-304/01 Reemergence of "=" in generics (8-0-0)
AI-324/01 Physical Units Checking (10-0-0)
AI-325/01 Anonymous access types as function result types (8-1-1)
AI-342/00 Requirement for freezing task storage on unchecked deallocation (9-0-0)
AI-358/01 Application-defined scheduling (8-0-2)
AI-367/00 Include type declarations for Natural and Positive in package Interfaces (10-0-0)
AI-368/01 Restrictions for obsolescent features (10-0-0)
AI-369/00 Completions and renaming (8-0-0)
AI-371/01 New hierarchy for OS-Dependent services (6-1-3)
AI-372/01 Restrictions for default stream attributes of elementary types (10-0-0)
AI-374/01 Assertions_Only pragma (8-0-0)
AI-375/01 Type and package invariants (8-0-0)

The following AIs should be merged into other AIs (thus becoming 'deleted' AIs):

AI-219/01 Conversions between related types (AI-251)
AI-336/00 Generic formal access types in Pure packages (AI-366)

The following dormant AIs were discussed briefly and (re)assigned:

AI-51/10 Size and Alignment Clauses for Objects
AI-86/06 Passing generic formal packages with (<>)
AI-109/07 Size and Alignment Clauses for Subtypes
AI-186/02 Range of Root_Integer
AI-226/01 Cyclic Elaboration Dependences
AI-239/01 Controlling inherited default expressions
AI-269/03 Generic formal objects can be static in the instance
AI-294/03 Instantiating with abstract operations
AI-303/01 Library-level requirement for interrupt handler objects
AI-309/00 Pragma Inline issues
AI-335/00 Stream attributes are [never] dispatching

## Detailed Minutes

### *Meeting Minutes*

The HTML version of the minutes says "21th meeting". The PDF version (which is the official version when there are differences) correctly says "21st meeting". The minutes of the 21st ARG meeting were approved by acclamation.

## Next Meeting

The 23rd meeting will be held in conjunction with Ada Europe in Palma de Mallorca, June 18-20, 2004. Tucker can't stay after Friday; he's going to Africa for a month. We don't want to meet without him. A suggestion is made to bookend the conference. That is approved (pending confirmation with the conference organizers). WG9 is on Friday, we decide to overlap some of the conference. The new dates are Sunday, Monday (June 13-14), Thursday (June 17).

Pascal would like to schedule an (extra) meeting in early September. SIGAda 2004 will be held in mid-November, which is quite late. Randy volunteers Madison as a meeting site. September is particularly nice in Wisconsin. But we have to check the football schedule; we don't want to try to meet on a football weekend. Tentatively, we'll choose Sept 10-12 for the meeting; Randy will try to find the football schedule over lunch.

After lunch, Randy reports that every weekend in September is a football weekend except Sept. 17-19. So that weekend is chosen by default. The meeting will be held Sept 17-19, 2004 in Madison, Wisconsin.

## Motions

The ARG thanks our host, Joyce Tokar, for her fine work on the local arrangements for the meeting, and the dining suggestions. Approved by acclamation.

## Old Action Items

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

Steve Baird:

- AI-251

John Barnes:

- AI-254

Randy Brukardt:

- AI-279
- AI-362
- Make a recommendation to the ARG about the alternatives for AI-302 (with Bob and Tucker).
- Send Alan Burns a summary of the editorial comments on AI-297.

Editorial changes only:

- AI-217-6
- AI-230
- AI-231
- AI-296
- AI-301
- AI-326
- AI-340

Alan Burns:

- AI-188
- AI-266-2
- AI-297
- AI-307
- AI-327
- AI-354

- AI-355
- AI-357

Gary Dismukes:

- AI-312
- AI-331

Bob Duff:

- Make a recommendation to the ARG about the alternatives for AI-302 (with Randy and Tucker).

Pascal Leroy:

- AI-285
- AI-315
- Check that the current wording of AI-326 properly handles all of the examples discussed in previous meetings for AI-217.

Erhard Ploedereder:

- Create an AI to allow access types with no storage pool in Pure units (see discussion of AI-362) [Assigned AI-366 after the meeting].
- Create an AI to add words to insure that it is clear that B.3 can be used to interface to C++ (see discussion of AI-285). [Now AI-376].

Jean-Pierre Rosen:

- Create an AI on a new restriction No_Obsolescent_Features (now AI-368).

Tucker Taft:

- AI-286 (split Assertions_Only into a separate AI - now AI-374).
- AI-288 (split into two AIs: pre/postconditions and invariants, now AI-375).
- AI-290
- AI-318
- AI-329
- AI-344
- AI-345
- AI-363
- AI-364
- Make a recommendation to the ARG about the alternatives for AI-302 (with Bob and Randy).

### *New Action Items*

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

Steve Baird:

- AI-51 (before April 1st)
- AI-109 (before April 1st)
- AI-294 (with Tucker and Pascal)
- AI-373

Randy Brukardt:

- AI-302 (before April 1st, with Matt Heaney and review of Tucker Taft and Bob Duff)

- AI-303
- AI-309
- AI-362
- AI-368
- Update Janus/Ada implementation report for current version of AI-217-6.

Editorial changes only:

- AI-148
- AI-209
- AI-245
- AI-251
- AI-286
- AI-329
- AI-351
- AI-360

Gary Dismukes:

- AI-269

Pascal Leroy:

- AI-100
- AI-186
- AI-226
- AI-239
- AI-294 (with Steve and Tucker)
- AI-335
- AI-359 (possibility 2, see discussion)
- AI-367 (to explain why there is No Action)
- AI-370
- Update Apex implementation report for current version of AI-217-6.

Erhard Ploedereder:

- AI-366

Ed Schonberg:

- Create an implementation report for AI-318.

Tucker Taft:

- AI-86
- AI-294 (with Steve and Pascal)
- AI-318
- AI-344
- AI-345 (before April 1st)
- AI-359 (possibility 1.5, see discussion)
- AI-363
- AI-364

- Review AI-351 for any violations of the Tucker model of time.
- Update AdaMagic implementation report for current version of AI-217-6.
- Create an AI on the restriction No_Dependence (see the discussion of AI-353 in San Diego).

Items on hold:

- AI-275 (pending resolution of AI-363; if AI-363 is approved as currently planned, this should be deleted or voted no action)
- AI-284 (waiting for more keywords to be defined)
- AI-295 (pending resolution of AI-363; if AI-363 is approved as currently planned, this should be deleted or voted no action)
- AI-356 (waiting for AI-357; if AI-357 approved as it currently is, this should be voted No Action)

## *Dormant AIs*

The meeting reviewed all of the dormant AIs (those that have not been discussed or updated since September 1st, 2003). The review was conducted with a 5-minute time limit, and was not intended to be a technical discussion. Rather it was intended to determine whether the AI should be voted "No Action", approved as is, or (re)assigned to an ARG member for further action (in some cases, these AIs were discussed during the technical portion of the meeting). 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.

### *Review of Dormant Regular AIs*

#### AI-51/10 Size and Alignment Clauses for Objects

Pascal notes that the aliased stuff is in AI-291. That should go away from this AI. The rest has lousy wording. There isn't a major issue here in practice; so we could say that it is insufficiently broken.

Erhard: We could agree to disagree, and table it. Pascal disagrees: the problem is the lack of a reasonable wording. Steve Baird says that he will work on it, before April 1.

#### AI-58/08 Accessibility rules for Shared_Passive Packages

This is all implementation-defined, and there's only one implementation at the moment. Let's trust the implementer to do the right thing.

No Action: 9-0-0.

#### AI-86/06 Passing generic formal packages with (<>)

Tucker suggests that that this be glued to AI-317, Partial Parameter Lists for Formal Packages. The issues are similar to those raised by Erhard in the context of that AI (and later proven to be not a problem). Assigned to Tucker to either figure it out or add to other AI. In particular, check if this AI is really a confirmation.

#### AI-100/03 Truncation required for static expressions if Machine_Rounds is False

AI-268 covers similar ground. It makes it more likely that Machine_Rounds is True for most targets. So it is OK to require truncation.

Approve AI: 9-0-0.

[Editor's note: After the meeting, it was discovered that meeting 8 had identified a problem with 4.9(38) which was intended to be fixed by this AI. Pascal Leroy has graciously volunteered to take the AI and update it as needed.]

### AI-109/07 Size and Alignment Clauses for Subtypes

See AI-51 discussion; Steve will send this one as well by April 1.

### AI-148/01 Requeue of protected entry calls

Add “? (No.)” to the question. Get extra space out of “a  call”.

Erhard says that “no” doesn’t reflect the response. He doesn’t know what the answer ought to be.

Tucker says that the question should say “protected action”, not “protected operation“; then the answer is clearly “Yes”. Randy points out that the original e-mail asks about “protected action”s, so the question is just wrong.

Approve AI with changes: 5-0-4.

### AI-172/04 Questions about main subprograms

John would prefer to change “any view of” to “all views”. Gary suggests “arbitrary views”.

Tucker notes that the significant issue is whether an implementation is required to support a rename of a subprogram as a main subprogram. The answer given is that it is implementation-defined; there isn’t much benefit to the ARG saying that here. And there is some risk of requiring something that we really don’t want to require.

No Action: 9-0-0.

John and Erhard ask that the summary be updated to say “arbitrary views”, to avoid confusion in future readers.

### AI-186/02 Range of Root_Integer

Typos: “of a the type”, “evalutuation”.

Tucker argues that we don’t say what happens outside of the base range. So we don’t require raising Constraint_Error. But it is allowed.

However, that makes instantiating a generic formal discrete type with the largest modular type a dicey proposition — many innocent expressions would be allowed to raise Constraint_Error. The AI is attempting to define the cases where that cannot happen.

Pascal will take this AI and search for a better solution.

### AI-187/00 Task attribute operations are atomic but not sequential

No work has ever been done on this AI, it can’t be important.

No Action: 9-0-0.

### AI-191/02 Are formal parameters passed by reference objects or views?

There is no answer in this AI. It’s never been updated (the version change was to correct the title). Steve comments that the manual suggests that the parameter is a new object. Tucker thinks that the wording could be improved.

Is there any case where this would matter? Pascal thinks that it might matter for pragma Atomic. No, that’s wrong, it doesn’t matter there. We cannot think of any reason that it would matter which way this is resolved.

No Action: 9-0-0.

### AI-200/01 Generic formal subprograms as dispatching operations

Pascal can’t believe that anyone does anything else.

Erhard says that he has just spent 10 hours discussing this issue. (Post-meeting note: And it came up again on Ada-Comment right after the meeting.)

Approve AI: 9-0-0.

### AI-209/02 pragma Reviewable; can objects become uninitialized?

John suggests “!status boring” for this one. Erhard says that it is fixing a real hole. He thinks that he wrote this AI.

For the wording changes, “usage of the value” isn’t right; say “read”. (If the object is being written to, we don’t care about its previous initialization state — that’s the point of the change.)

Approve with changes: 9-0-0.

### AI-219/01 Conversions between related types

Pascal thinks that this is subsumed by AI-251. So merge this into AI-251, and handle it there. (That makes the action Delete.) Erhard says that the replacement paragraphs in AI-251 still have this problem. We’ll discuss that in the context of that AI.

### AI-226/01 Cyclic Elaboration Dependences

Pascal thinks that the AI is fine as it is. Tucker thinks that other partitions should be allowed to continue. Erhard thinks that the Standard wording is confusing. Tucker thinks that the language is OK as it is, and supports his interpretation. If the language is to be changed, Tucker believes that Communication_Error should be a possible outcome (the partition may be talking to a partition that is not listening).

Pascal will take the AI. John suggests fixing the grammar mistakes.

### AI-239/01 Controlling inherited default expressions

Bob did not do this. This is an important safety issue in OOP programs. Pascal will take this AI.

### AI-244/01 Legal units which depend on illegal units

Pascal suggests that the Standard is correct as it is. No one disagrees.

No Action 9-0-0.

### AI-245/01 Consistency of inlined calls

John: “via a{n} automatic” in the summary.

Approve AI with change: 8-0-1.

### AI-269/03 Generic formal objects can be static in the instance

The minutes of the October 2002 meeting explain what this should do. Tucker mentions that legality rules are not enforced in instance bodies.

Gary will take this AI.

### AI-294/03 Instantiating with abstract operations

Someone thinks this should be a Binding Interpretation because it has wording. Randy explains that this could be a ramification because this is not a normative change (it’s in parenthesis).

Tucker doesn’t think this change improves the understanding of the Standard. Pascal, Steve, and Tucker will try to work out wording for future discussion.

### AI-303/01 Library-level requirement for interrupt handler objects

What we’ve done in new packages (the “Alan packages”) is to allow nested objects (defined properly), and then define a restriction to require all objects to be at library-level for efficient runtimes.

After discussion, we decide to adopt this here as well. Randy will rewrite the AI this way.

### AI-309/00 Pragma Inline issues

Pascal says most implementations allow this now, even though it is illegal.

Tucker suggests an Implementation Permission that the implementation can allow it elsewhere (for Ada 83 compatibility). The alternative, changing the rules for program unit pragmas, is unappealing because it could have nasty effects elsewhere.

### AI-335/00 Stream attributes are [never] dispatching

This issue needs to be resolved.

Mr. Stream (Pascal) will take this AI.

### AI-336/00 Generic formal access types in Pure packages

Subsume this into AI-366.

### AI-342/00 Requirement for freezing task storage on unchecked deallocation

The tasks may be components of a larger allocated structure — say a record or array component, but the wording doesn’t appear to cover that. That’s arguable; the wording doesn’t say “directly designated”. It doesn’t seem worth it to clarify this, why would anyone think anything else?

No Action 9-0-0.

## *Review of Dormant Amendment AIs*

### AI-218-01/06 Accidental Overloading while Overriding
### AI-218-02/04 Accidental Overloading while Overriding

These AIs are superceded by AI-218-03 (which has already been approved).

No Action: 10-0-0.

### AI-222/01 Pragma Feature

Tucker says that he still thinks this is a good idea. He seems to be alone in strongly supporting this idea.

No Action: 10-0-0.

### AI-232/01 Dispatching operation visibility and ambiguity

Subsumed by AI-252. Erhard would like to look at this to insure that AI-252 solves the problem. He is asked to do so before the end of the meeting.

On Saturday, Erhard reports that AI-252 does indeed solve the problem, in combination with AI-230.

No Action 10-0-0.

### AI-250/03 Protected types, Extensible, Tagged, Abstract

AI-345 seems to be a preferable way to solve this need.

No Action: 10-0-0.

### AI-266-01/05 Task Termination procedure

This proposal has been superceded by AI-266-02.

No Action: 10-0-0.

### AI-292/00 Sockets operations

Ed says some work occurred on this, but this was not finished. There is no proposal to work on, so it gets removed from the agenda.

No Action: 10-0-0.

### AI-302-01/07 Data structure components for Ada
### AI-302-02/02 Container library

We’re only going to work on the committee proposal, AI-302-03.

No Action: 10-0-0.

### AI-324/01 Physical Units Checking

Tucker reports that his proposal proved to be too hard to use. A more complex solution is needed, and he thinks it would require too much effort to complete.

No Action: 10-0-0.

### AI-325/01 Anonymous access types as function result types

This AI is closely related to AI-318, so we'll discuss it at the same time.

### AI-358/01 Application-defined scheduling

Alan didn't seem to think this a good idea; it goes beyond the state of the art and thus is premature. John says this is probably the fall guy for the real-time proposals.

No action: 8-0-2.

### AI-359/02 Deferring freezing of a generic instantiation

Tucker would like to discuss this. Randy thinks that it just got lost from the agenda because it was sent before a meeting. We decide to move it to the regular agenda.

## *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-251/11 Abstract interfaces to provide multiple inheritance

Tucker points out the need to drop the words "and is not an interface type (see 3.9.4)" from 3.4.1(10). The minutes failed to say this, but it was implied.

Also, the AI needs to be changed to incorporate AI-219 (as previously discussed).

The title of 12.5.1 in the wording is still wrong. "Formal Private and Derived Types". This was mentioned in the minutes from the San Diego meeting.

Add an AARM Note: To Be Honest as proposed in the San Diego minutes.

Steve will try to update this tonight.

On Saturday, Steve reports on integrating AI-219. Rephrase the bullets as a condition.

"At least one of the following conditions must hold:"

Tucker suggests "if the conversion is not permitted by one of the above rules, then the following…". Steve argues that that is a "notwithstanding rule".

Pascal suggests putting the common ancestor rule first. Tucker suggests putting 4.6(21-23) first, then say "otherwise:" Steve will try again tonight.

On Sunday, we take this up again.

Rewrite of 4.6: 4.6(24) becomes the first sentence of the legality rules section.

Steve reads the wording.

> In a view conversion for an untagged type, the target type shall be convertible (back) to the operand type.
>
> If there is a type that is an ancestor of both the target type and the operand type, then
>
> - The target type shall be untagged; or
> - The operand type shall be covered by or descended from the target type; or
> - The operand type shall be a class-wide type that covers the target type; or
> - The operand and target types shall both be class-wide types and the specific type associated with at least one of them shall be an interface type.
>
> Otherwise
>
> - [existing paragraph 8]
> - [existing paragraphs 9-12.1/1]
> - [existing paragraphs 13-17]
> - [existing paragraphs 18-20]

“If there is no common ancestor, then:” is better than “Otherwise”. [Editor's note: I actually used the inverse of the first sentence: "If there no type that is an ancestor of both..." because I don't think we want to have to define what "common ancestor" means.]

Approve AI with changes: 7-0-1.

### AI-286/05 Assert pragma

Tucker tells us that the main change is to specify a location for this pragma (11.4.2).

Why is the location specified? You don’t want it in the middle of record components or a generic formal part. “at {a}[the] place”. Inspection_Point (which is similar) uses “declarative_item” instead of “basic_declaration”.

Tucker wonders if there is an interaction with Suppress. Does All_Checks suppress assertion checks? Is there a check name? There should be check names for anything that can be suppressed.

Pascal thinks that it is a bad idea for Suppress to suppress assertions. We have a pragma for doing that. Tucker says that we can’t stop it — there is implementation-defined semantics allowed for Suppress. OK, but we don’t want implementations to do that. It is best to not talk about it.

If a different policy is in effect for an inlined subprogram (at the call) or generic unit (at the instantiation), what happens? Can we borrow the wording from Suppress? We look at the AI for Unsuppress, but unfortunately, we can’t. That wording depends on the fact that you don’t have to do the suppression.

Tucker would prefer to say implementation-defined. Pascal objects; you could be depending on the assertion happening. Pascal would rather that the policy for the body (generic or inlined body) applies in the instance or call.

Tucker would prefer that the default policy can be overridden. That’s OK, but if the user specifies something, it should hold. We agree on that. We’ll need to say something to make this true. “If the Assert appears within a generic unit, the policy within the generic unit determines the policy which applies, regardless of the policy in effect at the location of the instance.” Ugh. Try: “The assertion policy that applies within an instance is the policy that applies within the generic unit”. “AARM Note: The assertion policy that applies within an inlined body is the policy in effect in the body, because inlining does not affect dynamic semantics.”Change the equivalence to “raise Ada.Exceptions.Assertion_Error with Message”. That uses the much clearer syntax of AI-361. Besides, Pascal points out that, as usual, the equivalence doesn’t quite hold, because a call to Assert always evaluates the Message.

John comments that the implementation-defined policy is confused with the default policy. Indeed, this isn't organized like the other policies. Describe the policies with words, not grammar; that seems clearer. And it just can be identified as a *policy*_identifier.

Pascal wonders about the Note. Steve thinks it isn't true. They use caching in Assertions, for instance. That's OK, but anything significant would cause trouble. Change to "significant side effects". Move "Normally" to be the first word of the sentence.

Approve AI with changes: 7-0-1.

### AI-288/02 Pre- and Postconditions

Tucker updated the AI by splitting Invariants as directed. He moves No Action. He doesn't think that we can finish the needed work on this proposal in time to meet the schedule.

No Action: 8-0-0.

### AI-290/01 Declaring functions Pure

This pragma differs from the existing categorization pragmas: Pure is checked, whereas this is an assertion that the function is Pure. The optimization permission is the same as for Pure: the function can be called only once and the result reused; but the compiler is not allowed to call the function twice and assume that the result doesn't change.

Erhard wonders if there is any possibility of optimization here for access parameters. If not, we shouldn't bother writing so much text to explain when it is allowed. Tucker and Randy say yes, at least for access to object.

Pascal says that this is only for optimization, and he worries about the fact that this isn't checked. That means that it is really easy to shoot yourself in the foot. He would like to see some checks done in the function body, but the rules would be really hard to write, or they would be so restrictive as to make the pragma useless. Steve wonders if the safety issue is worse than Suppress. Ed says that this is an assertion that the compiler can't figure out. A compiler could have warnings for unsafe use. Pascal argues that if optimization is the only benefit, the pragma might as well be an implementation-defined one.

Steve comments that IBM Rational would probably not use it; you could get a lot of the benefit from using pragma Inline.

No Action: 9-0-1.

### AI-302-03/02 Container library

Pascal would like to make a list of issues. This seems like a reasonable way to proceed.

1. Do we want this, and is the scope (and size) of the proposal OK?
2. Lists?
3. Performance issue, what should be specified or not?
4. Indefinite elements?
5. Case insensitive comparisons broken? More hash functions/fixed/unbounded?
6. Non-library level instantiations?
7. Iteration mechanisms? (Passive iterator for vectors?)
8. Aliasing of elements (which constrains object); pointer to elements?

9. Array sort?
10. Signature packages?
11. Size_Type thing, Hash_Type range?
12. Names of packages/types?
13. Indexes are signed integer vs. any discrete?
14. Empty space?
15. Set-oriented operations: union, intersection, etc?

**Do we want this?**

Yes, the group agrees. We list the names of the units in the packages for reference.

Ada.Containers
Ada.Containers.Vectors
Ada.Containers.Hashed_Maps
Ada.Containers.String_Hash_Maps
Ada.Containers.Wide_String_Hashed_Maps
Ada.Containers.Sorted_Sets

Ada.Strings.Case_Insensitive
Ada.Strings.Hash
Ada.Strings.Unbounded.Hash

**Should we include Lists?**

Randy says that it depends on the performance specs. List and Vectors have similar capabilities in the absence of performance requirements. Ed says that he thinks that it makes the library complete from a beginner's perspective. Tucker thinks it is not important enough. Pascal agrees with Randy. Matt says that he uses list for queues.

We decide to discuss performance issues and come back to this.

**How tightly should performance be specified?**

One view is that they should be specified tightly, so that the performance is well specified. The other view is for loose performance bounds.

Pascal would like a looser specification that eliminates stupid things like using Bubble Sort. The constant factor is not specified; it could be huge. That could overwhelm the order of N factor.

Pascal wants O(log N) rather than O(1) for insertions into vectors. He argues that in practice you can't tell: for problems that are so big that the difference is significant, caching effects will in fact dominate. Similarly operations that are specified as O(N) should really say O(N log N). Sorting currently is O(N log N) (on average). Pascal would like to say less than quadratic, for instance to allow Shell sort.

He also argues that implementations might want to compete on performance, and we should not hamper this by overspecifying the behavior.

After some discussion the group agrees.

**Returning to lists:**

Ed says that from a marketing perspective, lists are necessary.

We take a straw poll — Should we add a list component? 6-0-4.

The list component should be a doubly linked list to allow reasonably efficient insertions and deletion. Name it Doubly_Linked_Lists. (Matt proposes Double_Lists, but saving a few characters doesn't get much support from the group)

**Should we support indefinite elements?**

Yes, name them Indefinite_Vectors, Indefinite_Hashed_Maps, Indefinite_Sorted_Sets. There is some discussion regarding maps: do we want both Key and Element to be indefinite? Matt doesn't think so. Pascal believes that we want to give the user full flexibility, so allow both types to be indefinite. The element and key types have to be non-limited. Define the text of these packages by substitution.

The wrapper package idea that was mentioned in ada-comment discussions is thought to be more complicated.

Should the string ones then be removed? The instantiations of the indefinite forms are nearly as easy. Matt would like to keep them, because the string version can depend on a discriminant, and avoid the heap use. The functions have useful defaults as well.

Matt doesn't get much support. The string routines are removed.

**Should we have case insensitive comparisons, and if so, what should they look like?**

Pascal wonders why they are there. Tucker points out that you want to provide standard hash functions and comparison functions —and case insensitive sorts are common.

Tucker points out the names of the operations in Case_Insensitive don't work. We could have a type Case_Insensitive_String is new String. Yuck, that would require lot of type conversions. Otherwise, we could have names:

Case_Insensitive.Less_Than ()
Case_Insensitive.Equal ()
Case_Insensitive.Hash ()

Pascal thinks that this should be done for all of the string packages. The string packages already support case insensitive operations with Map parameters. We don't need to provide duplicate functionality. So if we do this, we should support it more generally. But that is messy. These are easy to write when they are needed to terms of Maps. So drop the entire case insensitive stuff; users can call To_Upper if they need to. Matt disagrees, this is common enough to support specially. He doesn't change anyone's view.

Bounded hash would have to be a child of a generic; which would require an extra instantiation. Pascal again says that we should be consistent. Randy points out that we didn't provide a bounded Text_IO either, so we are being consistent.

**Should we require non-library level instantiations?**

Non-library level instantiations would be nice. We don't want to require anything that can't be written in Ada. So we need to make it possible to write these packages in Ada. That is a separate problem, to be solved in AI-344.

**Are the iteration mechanisms what we want?**

Pascal hates Front and Back. So does Randy.

Matt writes examples of active iterators:

Set:

```
S : Set_Type;
```

```
   I : Cursor_Type := First(S);
   J : Cursor_Type := Back(S);

   while I/= J loop
      E := Element (S);
      Increment (I);
   end loop;
```

List is the same as Set.

Vector:

```
   for I in First(V) .. Last(V) loop
      E := Element(V,I);
   end loop;
```

Map:

Same as Set, except Increment has the container as an extra parameter.

Ed and Pascal dislike showing the sentinel node. Why don't you just stop on Last? It's intended to look like the STL.

Matt explains that this mechanism works on subranges as well. Pascal thinks that subranges are unlikely, Matt thinks they're common.

Tucker says that this is modeled on STL. We could add operations. But we should stick with this design. Pascal wonders why; he thinks it's ugly and flawed, and why do we need to mimic the mistakes done by C++?

Pascal would prefer an iterator object. Randy wonders how that would work. Tucker explains that this would work with a Next/More/Get (or Next&Get together).

```
   while More(It) loop
      X := Get_Next(It);
   end loop;
```

Cursors remain valid when you do modifications, with the exception of the item directly pointed to. Iterator objects would have to have a check or something.

Matt notes that the existing scheme is exactly the way lists are iterated in Ada. Some suggestions are made to change the name of Back to End_Set or Sentinel. Pascal says he still dislikes the scheme altogether. We seem to have three options:

1. Keep as is, cursor approach;

2. Switch to iterator approach;

3. Have both.

Randy points out that you can compare against Null_Cursor. Why do we need a separate Back routine? Matt says that the sentinel is an artifact of the container. Pascal says (again) that the sentinel shouldn't be exposed.

```
   while I /= Null_Cursor loop
```

This *is* exactly how lists are iterated in Ada.

Ed says that Matt is too focused on the sorted set. He wants a more general abstraction. We don't need to have Front and Back then.

Tucker does not want to explicitly test for **null**. He'd rather have something like a function More.

```
   while More(I) loop
```

Matt objects. He doesn't understand how this would work. The cursor type would have to know whether it is at the end of a container. Matt says that that makes the cursor bigger. That's OK, there are not many cursor objects in a program.

Matt still doesn't like this. What does Find return? Null, of course. Matt doesn't want to do that.

Tucker suggests changing the name of More to Exists, which would address this concern.

The wording should say that cursors should get initialized to something for which Exists is False.

Let's summarize the active iterator discussion. Functions Front, Back, and constant Null_Cursor are eliminated. A function Exists is added. Change Increment and Decrement to Next and Previous. Do we need the function version of Increment and Decrement? About half want the function, and half want the procedure. So we leave both — with the names Next and Previous.

Tucker would like a combined version of Get and Next. We clearly need the separate routines. We could add a procedure that does both. Tucker realizes that a procedure wouldn't work with indefinite, and you'd still have to do the separate operations. So forget this suggestion.

**What passive iterators should we have?**

Matt again shows an example of use:

```
declare
   procedure Process (C : Cursor_Type) is
      E : Element_Type := Element(C);
   begin
      ...
   end Process;

   procedure Iterator is new Generic_Iteration (Process);

begin
   Iterate (S);
end;
```

Tucker asks why these iterators don't take elements rather than cursors.

For vectors, we don't have cursors. Matt shows how it works.

```
declare
   procedure Process (E : in out Element_Type) is
   begin
      ...
   end Process;

   procedure Iterate is Generic_Iteration (Process);

begin
   Iterate (V);
end;
```

For Maps, using elements gets complex.

```
procedure Process (K : in Key_Type; E : in out Element_Type) is
```

You would like to have all of the combinations. Just having the cursor allows to do all of the things.

The problem for Vector is that you really need four of them (all combinations of in/in out, forward/back).

Tucker thinks that the passive iterators for vectors are silly — they're not consistent, and consistent ones on Index_Subtype are useless.

Pascal points out that a uniform solution could be used for a generic formal package.

Tucker would rather have a simple version that always gives the element, and offer the cursor version as an extra.

Perhaps the vector should have a cursor type that is not fragile. The cursor would usually be the index, and a pointer to the object. Cursor cannot break for insertions. That has to be said for all of these packages.

Summary: add cursors and consistent passive iterators for vectors.

**Why do we need pointers to elements?**

Tucker thinks that if we have cursors, we don't need them. After all, a cursor is a more abstract way of designating an element.

Matt says that it is important when composing containers. You don't want to copy the whole element to just to add one item to a contained list, for instance.

Tucker suggests having a generic for doing modifications, which gets rid of the pointers.

```
generic
   with procedure Update (E : in out Element_Type);
procedure Update_Element (Cursor : in Cursor_Type);
```

For the definite versions, it should be required that elements are never constrained if Element_Type is unconstrained. The implication is that you can't directly alias a component.

We could use a return by reference function (however that is handled in AI-318), which would allow renaming. But we'd like to stay with Ada 95 in order to get this in use early. So the generic is preferable.

**Should there be a sort for array?**

The algorithm would be the same for arrays and vectors, and it seems silly to force users to create a vector when all they want to do is sort an array. Add Ada.Containers.Generic_Array_Sort. Note that there is no reason to assume that the *implementations* are shared.

Should this be a child? Randy dislikes children for both implementation and stylistic reasons. Erhard would like the possibility of other implementations, which need a sort. That argues for a child. But that's akin to overspecifying the performance: most users will want a "good" sort, without having to select the algorithm themselves. So stick to Ada.Containers.Generic_Array_Sort.

We don't want a stable sort, because it takes more space or a lot more time. You can always make a fast, unstable sort stable with an extra original position component.

We want to say that the average performance is better than O(N**2).

Is the formal array type unconstrained or constrained? Why is constrained important? Because you may not have an unconstrained first subtype, and that may not be possible to change that. Tucker thinks that never happens, but Ed and Pascal say that's a real problem.

Steve suggests that one constrained version is enough. You could use a subtype to sort a slice.

We eventually decide that we should have both. Generic_Array_Sort, Generic_Constrained_Array_Sort.

Tucker would like to leave out the "Generic". We look at the existing predefined units, and half of them say "Generic", half of them don't. We decide to keep "Generic".

**Should we use signature packages?**

Tucker tries to explain how these would be used:

```
generic
   type Set is private;
   with procedure Insert ();
   with procedure Delete ();
package Set_Signature is end;
```

Then this is instantiated with each set abstraction. (This could be done inside the package with some rule changes.)

```
package Set_Sig is new Set_Signature (My_Set.Set_Type);
```

Then you can use "`with package Set is Set_Signature (<>);`" to create a layered abstraction.

Tucker doesn't think this is worth doing if it can't be part of the initial container abstractions.

Steve suggests a skin package:

```
generic
   with Set_Package (<>)
...
```

Randy wonders why you need a signature package; at this point there is only one kind of set. It seems that it belongs in a secondary standard. Pascal says it provides a guideline for how to do it, and having it there now would avoid name problems with adding it later.

We decide to postpone a decision until we have discussed AI-359. But since the discussion of AI-359 was itself inconclusive (see below), we have to wait until we have firm proposals for solving the interactions between generics and freezing.

**Size_Type is declared in Containers. Is this right?**

Our choices for this are:

1) Use Natural instead of a specific Size_Type;

2) Size_Type in containers;

3) A specific type in each package.

Size_Type is used as the value of Length, and for preallocation. Bob had said that size should be a subtype of Index_Type'Base for vectors. But that doesn't work elsewhere. And you're not going to use it for indexing.

We think that the AI has this right.

What about the upper bound of this? There isn't much support for specifying a lower bound on the upper bound. Just remove that.

The hash bounds have similar arguments and a similar conclusion.

Should there be implementation advice that these "should normally be at least 32-bits"? That seems like a good idea, because the users need to have some idea about the range of the types.

**What should the names of the packages be?**

Pascal does not like Vector, because of confusion with AI-296 type Vector. Tucker points out that both C++ and Java use Vector for this. Pascal would like Sequence. But both Vector and List are Sequences. Vector is chosen.

Why is the package named Hashed_Map? It provides a hint about the capabilities; it requires a hash function to instantiate. It is a particular kind of map. Pascal would prefer it to just be Map, but he loses that battle. Ed suggests being consistent, the set package should be named Ordered_Set. After much discussion, general agreement is reached on this.

**What should the type names be?**

Currently, we have Vector_Type, Set_Type, Map_Type, List_Type for the container types. And we have Element_Type, Cursor_Type, Key_Type for the ancillary types.

Choices are:

1) As Is;

2) Drop "_Type" everywhere, but that interferes with names of functions;

3) Drop "_Type" on declared types, keep on formal types;

4) Use same name for main type e.g. "Container".

We take a straw poll. Is this an acceptable (unacceptable) choice: (1) 6-2; (2) 4-3; (3) 6-3; (4) 2-8. We try to narrow this down by a second vote: prefer (1) 4; (3) 4; abstain 2 (unsurprisingly, Matt would leave it "As Is").

Matt used the file I/O packages as a model.

Tucker suggests naming the formal parameters Source, Target, From, etc. Or possibly, use "Container" as the formal parameter. Use same reasonable name for formal parameters across packages.

Take another vote: prefer (1) 3; prefer (3) 7. (3) sort of wins.

**Should be the index for Vectors be a signed integer or a discrete?**

If you have a type with 'Base'First='First, then it is going to fail (an empty Vector has Last(V) = 'Pred('First).

Why not use Integer? It might not have the required range. And of course, it isn't strongly typed.

We take another straw poll:

1) range <> gets 4 votes;

2) (<>) gets 3;

3) don't care gets 2.

So this is vastly inconclusive.

**Should there be an operation for inserting empty space in a vector?**

Matt explains that if you know you're going to insert M elements, using Insert (Vect, Index, Element); to make single inserts would take time O(N*M). But Insert_Empty_Space (Vect, Index, Size); would take time O(N).

The current semantics are that the elements are uninitialized (what ever is there is there, or default initialized if new). Someone suggests that the name should be "Insert_Hole". We do not want to specify what is in the new elements (that's especially important for the indefinite versions). "Open_Up" is suggested as a name.

We decide that this is useful, and stick to Insert_Space as the name.

**What should the Delete operations look like?**

Some of the Delete operations are a bit weird. Adding and removing from the end are special. Matt wonders if the multiple delete should be Delete (Vec, First, Last) or Delete (Vec, Index, Size). The latter is more consistent.

Pascal wonders why Delete_Last is important. Matt says that it is the analog of Append. Pascal says that then he would want Delete_First. Matt says that this is expensive, and you can write it yourself. Delete_First could be implemented efficiently. Then we should have a Prepend and Delete_First. That would be consistent with List; so those routines are probably a good idea.

Insert, Prepend, and Append of vectors also. Delete, Delete_First and Delete_Last all including a Size parameter, which defaults to 1.

**What about Swap?**

Pascal asks why there is the odd Swap operation. Matt explains that it provides a form of move (for large containers). Move might be easier to explain; it is an assignment, with the Source empty afterwards.

Replace Swap with Move, the Target is required to be empty beforehand, the Source is empty afterwards.

Swap (Index1, Index2 : Index_Type) is an open issue. Swap of elements would be more efficient when a level of indirection is used (e.g., for indefinite elements), so that's useful. We also want all these operations with cursors as parameters (remember that we added cursors to vectors).

**Other operations for the containers?**

Should we have concatenation for Vectors? We should have the usual overloads of "&" for Vectors.

Operations that are common between the containers are cursor oriented. We want these to be as similar as possible. Add commonality in the discussion of the AI.

Should we have set operations for Set? Union, Intersection, Difference should be added. We already have membership with Is_In. These should have procedure and function forms. Procedure has **in out** first, **in** second. For function, also provide renames of Union as "+", Intersection as "*", Difference as "-". Other operations: Complement makes no sense, we don't know the universe of elements. Pascal wonders if "and" for Intersection and "or" for Union would be better. Function symmetric_difference is "xor". Forget "+" and "*".

Matt asks for a preinstantiation of Text_IO for Size_Type. Everyone tells him to use Size_Type'Image.

What does Lower_Bound do? It returns a value near a value; sort of related to Find. Find is an exact match. Tucker says this name doesn't grab him. The semantics is nearest above or equal for Lower_Bound. Upper_Bound nearest above (not equal). UB >= LB; UB > K; LB >= K. Floor = Upper_Bound - 1. Ceiling = Lower_Bound. These come from STL. The group looks at the definition of these notions, and gets a collective headache. What the hell is this useful for?

You can give it as a hint as to where the ordering should go. Get rid of the hint. (Insert with position also would be removed.) Get rid of the Upper_Bound and Lower_Bound routines. Perhaps we should have Floor, Ceiling to replace them. No, that seems insufficiently important.

Ed asks if we should have Is_Subset (A, B : Set) meaning is A is a subset of B. Yes. Empty_Set is a constant. Probably should be an Empty_Something for all containers. Is_Disjoint (A, B : Set) means the intersection is not empty. But "not Is_Disjoint" is awful. Tucker says they use "Overlaps" in their projects. John thinks that this disgusting. No conclusion is reached.

Should there be more functions for Map? The Set ones don't make sense. We can't think of any routines that need to be added for Maps.

The committee will produce another draft by April 1st.

Approve intent of AI: 8-0-2.

### AI-304/01 Reemergence of "=" in generics

Tucker explains that in Ada, tagged types work "right". (That is, equality composes automatically.) It would be nice if there would be untagged types that worked "right". Thus he invented **aliased** types.

Pascal says that this is a new type that buys you a few extra properties that have nothing to do with the main feature. That means that you have to specify **aliased** when you don't mean it, and that's bad. That's precisely the problem we're trying to fix for **limited**, it would be nasty to re-introduce it here.

Randy wonders what happened to the resolution of this problem agreed to in Burlington. Tucker says that using a pragma or attribute to specify this didn't pass the "bolted-on" test. (That is, such a solution would look bolted-on to the language forevermore; it would never look like a designed part of the language.) Thus, he turned to this solution.

Tucker moves No Action; he thinks it does not directly solve the problem, and when it is used to solve the problem, it triggers other problems.

No Action: 8-0-0.

## AI-318/03 Returning (limited) objects without copying

Tucker explains the changes. He added wording. The return statement part is a complete replacement for 6.5.

The AARM Ramification wording needs a period.

Steve points out that there is an obscure case where adding aliased won't work (if an access-to-subprogram points at both return-by-reference and regular functions). Tucker thinks that that is a good thing — the semantics of the functions are completely different, and they really can't be used in the same way.

The extended return works like an accept body; it includes a handler.

John wonders about "return expression" — was it intended to cover both kinds of return? Yes.

The extended object can be constrained by its initial value; that ought to be said.

Pascal says that he finds the model of aliased functions hard to understand. Tucker explains that this is a way to return a reference; this is something that is common in other languages.

Pascal thinks that he would prefer AI-325's model for returning references (that is, anonymous access returns).

It would be possible to replace the aliased capability with anonymous access, but the limited object return is different. (It's about build-in-place, while anonymous access is about returning a new or existing object. Randy points out that you don't want to force users to use access types if they don't need to do so.

Returning anonymous access is a harder problem implementation-wise than aliased functions; it needs to pass in a storage pool as opposed to a buffer.

The existing concept of return-by-reference type is dropped by the proposal; we now have return-by-reference functions. When the text says 'limited' in the dynamic semantics; how is this determined? Usually, there is a meta-rule that you ignore privacy in dynamic semantics. This is a tagged type, so limitedness cannot change between the partial and full views.

Pascal asks how you check the constrained restriction in a generic. If you are returning a limited private formal, you don't know whether it is limited or constrained. You could use definite — no, that allows too much.

You could use the runtime check (raise Program_Error if the check fails). Yuck. Tucker wonders if the check was moved to the freezing point of the function, then you could assume-the-best in the generic body. The discriminants would not be visible in the body. Pascal is unconvinced. What about an unconstrained array of these functions?

Pascal still thinks that the aliased function stuff is too complicated. Tucker says that it could be replaced with anonymous access, but that requires changing the call site. So it's even less compatible than aliased functions.

Ed suggests that we simply allow a procedure call as an initializer, instead of all of this. He proposes a simple syntax change:

```
Obj : Lim_Type with Proc_Call(Obj);
```

Tucker doesn't believe that this solves the whole problem. In Sofcheck's projects, they would have liked to use Sets that were limited to control copying; but they couldn't do it because then they lost everything that wanted to do (such as Union functions).

Ed doesn't buy this. This seems to be a job for controlled types. Tucker says that all they wanted to do is to prevent copying. That's what limited is for. Controlled types are a lot more complicated.

Erhard asks if this is a way to get constructors. Tucker says that he thinks that functions are the constructors for Ada. This argument seems awfully familiar.

Pascal suggests allowing prefix notation here:

```
    … := Thing.Proc(…);
```

and simply allowing a procedure call as an initializer.

Volatile mentions return-by-reference type; that reference should be removed.

Steve asks if an extended-return is a master. Let's discuss that off line.

Tucker thinks that this is easier from the user point of view. Pascal thinks that this is harder to understand. Ed shares Pascal's concerns about the complication.

Pascal says the extended_return is not the problem. What bothers him is the aliased stuff. The problem is that we have to do something about return-by-reference, because doing the difference between return-by-reference and return-by-copy at runtime is awful. We could just blow return-by-reference away completely.

We turn back to Ed's idea: It would allow a procedure call in a declaration. But it wouldn't allow them in the other places that functions can be used; such as return, aggregates, allocators. Gary and Erhard agree that it isn't worth it without getting those places.

We list the various alternatives:

A. Limited function with extended_returns:
How to move the existing return-by-reference functions out of the way:

   1. eliminate feature (must use named access return);

   2. aliased return;

   3. anonymous access return.

B. Procedure call for initialization → keep "rotten" return-by-reference.

How bad is the upward incompatibility for each of these alternatives?

| | |
|---|---|
| A1 | Change all calls, declare general access type, change return type to general access type, change returns to '[Unchecked_]Access; |
| A2 | Add "aliased" to all affected functions; |
| A3 | Change all calls and add return "access", change implementation of function; |
| B | No incompatibility. |

We take a series of straw polls:

| | |
|---|---|
| A1 | acceptable: 5 not acceptable: 0; |
| A2 | acceptable: 5 not acceptable: 2; |
| A3 | acceptable: 7 not acceptable: 1; |

B acceptable: 2 not acceptable: 5.

We decide to proceed with A3). That is, anonymous access types in returns with restrictions to eliminate anything needing to come from the outside of the call. Allocators would go into the standard pool.

Tucker needs to write up a proposal. Make it an alternative to AI-318. We don’t need AI-325 in this case.

### AI-325/01 Anonymous access types as function result types

This AI will be folded into the AI-318 proposal(s).

No Action: 8-1-1.

### AI-329/03 pragma No_Return

Erhard says that there are more cases of subprograms not returning than mentioned in the summary. The summary is too specific. It needs to allow for infinite loop as well, or for another task aborting the current task.

There is a typo in Dynamic Semantics: “appies”.

Paragraph 1 of the wording is too specific. “it never returns normally.” Or “it can return only by propagating an exception.”

Add a comma at the end of the first line of the syntax.

Also in dynamic semantics, again “rather than propagating an exception.” That text should be deleted, we don't want to list all of the ways that something can be abnormally completed (that's defined in 7.6.1(2)).

Do we have to say this applies to all instances of the generic? No, that is covered by 10.1.5(7.2/1). But that is only implementation advice. So this should say something similar to what is said for Inline: see 6.3.2(5).

Approve AI with changes: 9-0-1.

### AI-344/01 Allow nested type extensions

Tucker explains that the rules on accessibility of tagged types are repealed, replaced by checks to avoid ‘leaking’ objects of types declared in inner scopes outwards.

Randy’s primary concern has been addressed by the retention of 3.9.1(4). This is necessary both to avoid contract model problems and to ease implementation. Tucker would like to weaken this rule to allow instantiations that happen to derive types. Say that, in a generic body (G1), you instantiate another generic (G2). This instantiation might well contain derived type declarations, e.g. in its private part. Tucker thinks this could be allowed because the situation is not as bad as the general issue addressed by 3.9.1(4). Some people see this as critical for usability of the containers: G1 might be written by a user who wants to declare a set by instantiating Ordered_Set (aka G2) and this instantiation is sure to declare derived types (controlled types).

Randy claims that these have the same contract model problems. Randy and Tucker will take this off-line.

Gary wonders if the statically checkable case is really possible to check that statically. There are currently runtime checks for 'Access in similar cases (because of contract and sharing issues). So this needs to be a runtime check.

Tucker says that his implementation has problem with functions with controlling results that don’t have any parameters of the type — in that case, you don’t have the static link or display. So he had to create and pass in a dummy object. This doesn’t seem to be a general problem.

Steve says that this makes T’Class’Input harder. You have the tag, but the tag doesn’t contain the static link (because it is created at compile time). You’re creating the object, so you can’t look there. You’d have to get the link from your current state.

Should Internal_Tag not allow these things? No, that would be a privacy violation. Could you require it to exist? No, for the reasons outlined in AI-279. We'll use the answers in that AI for Internal_Tag.

Returning to 'Class'Input. Tucker says that this tag would have to be on your dynamic chain; you have to have a dynamic look up. You would need a per-task data structure to find the appropriate information. That could be a pointer to something in the stack frame.

Steve wonders if streaming out the static link would fix the problem. No, that wouldn't work on later runs, which would be devastating for components.

The data structure Tucker suggests could be used in AI-279 to eliminate all of the erroneousness (you would have a way to find all of the active non-library-level tags for the current task, and we already have ways to fix the problems for library-level types).

Approve intent: 8-0-2.

Tucker will update the AI.

### AI-345/04 Protected and task interfaces

Tucker described the basic idea: protected and task types can have interfaces, and the compiler will arrange (probably with a wrapper) that they can be called.

Pascal notes the example is wrong because it has the old task interface syntax.

The use of null procedures shouldn't affect which operation was overriding. The wording has that effect; it differs outside the task type or inside it. Also, we must disallow overriding both inside and outside a task (can't have two implementations in the same tag).

Pascal speaks in favor of the proposal. It's a nice integration of tasking and protected types. Polymorphism, not inheritance, is the big win here.

Steve wonders how much programming effort this can save.

Ed says that he has never seen a convincing example. He also thinks it is a lot of work coming late in the game. Randy thinks that it is not a lot of work on top of interfaces (interfaces is the big job).

Gary also suggests that he would like to see a compelling example.

Tucker is asked to put the most interesting example (a thread-safe and a fast queue, using the same interface) first in the AI.

Tucker comments that this seems like a natural thing to have allowed, and it is missing in Ada. Java does support this sort of thing with protected types. Ed doesn't think that much interesting is done with that Java capability. Tucker thinks that Java users are used to this capability, and do use it.

Tucker comments that it might be best to just say that protected and task types are tagged. (Currently, it just says types with interfaces.)

Steve wonders if a derived task type could add interfaces (that would cause a small amount of task distributed overhead). Tuck says that is something that is unintentional if the proposal allows it.

Erhard wonders if you would want to have something else like Synchronized_Interface that would allow either a task or protected type. (The proposal includes similar predefined Task_Interface and Protected_Interface).

Ed thinks that implementing dispatching entry calls is a problem. Tucker has shown a possible way to implement that (in response to Randy's similar question in e-mail).

Erhard worries that the interface doesn't expose blocking versus non-blocking for a task-only interface. That seems to make analysis harder. Steve comments that that is a feature; you're abstracting that away in the interface.

Selected procedure calls have the semantics of the ‘entry’ being open. But we don’t allow that when it’s clear that the procedure is a procedure (that seems like a mistake); the called item would have to be allowed to be an entry.

Should you be able to ask for the tag of a task that implements an interface? The interface is a tagged type. But Tucker deleted that! That wouldn’t work. Removing “tagged” from 3.9.4(1) seems to be dangerous. If he really wants to do that, Tucker should do a thorough analysis of the impact on AI-251.

Straw vote: keeping the discussion going: 7-2-1.

The proposal should emphasize that this is about polymorphism.

Tucker asks whether we should have the magic interfaces. Erhard says that he’d rather have the synchronized version. Indeed, he’d rather have it in syntax:

```
    type identifier is [synchronized] interface;
```

Erhard wonders why would you want to know that something is a task? So it can be aborted, passed to Alan’s packages, etc.

Pascal agrees that he would prefer to have the restricted interfaces be declared with syntax.

Ed wonders if we could simplify this by only applying it to protected types. Pascal and Randy concur. Tucker disagrees; active objects are often discussed, and it would weird to stop halfway. Gary agrees with Tucker. Erhard suggests that you may need to convert from a PT to a task (because of blocking operations), and it would be bad to lose the capability to have interfaces when that happens.

Should entry families be included? Randy explains that it is just is a special procedure parameter for protected operations, and it would be weird to not be able to do that. Tucker points out that the implementation is easy. Erhard thinks that the use of the second parameter is weird. Straw vote to support them: 2-4-4. So we’ll drop them.

Tucker asks if a plain old task type could have 'Tag. That seems to be unnecessary. Randy comments that the task interface would allow a generic formal task type, and you might have a contract problem. True, but if you use a syntax to declare task-only interfaces, you don’t have a name to query, so tasks that don’t have interfaces really don’t need tags. The group finds that useful, and it argues in favor of syntax for the “magic” interfaces.

Ed asks if you can use 'Class on a task type. No, a task type is not tagged, 'Class is only allowed on interfaces. The idea is that 'Tag is only legal on types that implement interfaces.

A Type’Class where the type is a task interface or implements a task interface is a task type for abort, 'Identity, 'Terminated, 'Completed, etc.

We need combination rules to prevent defining unimplementable interfaces (“mule” interfaces). (We don’t want interfaces which require the implementation to be both a task type and a protected type.)

Pascal asks about membership tests. We can’t think of anything special there (everything is covered by AI-251).

Ed wonders about 'Count. That can only be used inside the entry, so it is unaffected.

Pascal wonders why you can’t call entries as a normal procedure rather than the prefix notation. That wouldn’t be upward compatible; it would flood the scope containing a task object with names that aren’t there currently. Pascal groans, and asks us all to forget that idea. Sorry, it’s already recorded in the minutes! Tucker points that you can call an entry as a normal procedure for types (and procedures) that have interfaces. Erhard points out that there is a minor hole in that AI-252 doesn’t apply to task and protected objects that implement interfaces, if the operations are declared outside of the type. That should be fixed.

Approve Intent: 7-2-1 (Gary and Ed vote against. They believe that there is insufficient benefit for the cost.)

Tucker will update the AI by April 1st.

## AI-351/03 Time Operations

Randy explains the changes. The original package was split into three packages with better names. The leap seconds operations were added.

The Calendar math certainly needs to include leap seconds. Difference should say Seconds + Leap_Seconds = Calendar."-" if Day_Count = 0. Other operators in Calendar should be consistent with that.

Typo: in the second line after the package specifications, “I” is missing from “implementation”.

Last paragraph (AARM Note) of “Subtracting Duration(Local_Time_[Zone]{Offset}*60) from Clock provides the UTC time. In the US, for example, {Local_}Time_Offset will generally be negative.”

Tucker disagrees with this paragraph. He says the time zone offset only affects Split. A long discussion ensues. The primary point was that there are two possible models of Calendar.Time: either it is an abstract time value (such as August 31, 1958 09:52:00) where any time base is implied by context; or it is a time value of a specific time base (such as UTC). There is no practical difference in implementation of these models, but in the latter model, time zone conversions can only occur on output (Split in this case), as subtracting an offset gives a different UTC time, not a value of the abstract time with a different (implied) time base.

We try to reword the offending sentence to fit into Tucker’s model.

“Calling Split on the results of subtracting … provides the hours/minutes/seconds of the UTC time.” The other parts of the date also can depend on the time zone. Use “provides the components (hours, minutes, and so on) of the UTC time.”

The Time_Offset is relative to the local time. Therefore, Local_Time_Offset should be called UTC_Time_Offset.

Delete the second sentence of the description of type Time_Offset, because it conflicts with Tucker’s model.

Tucker would prefer to call the “Time_Zone” parameter “Offset”. That doesn’t sound very descriptive. Descriptions should say “timezone offset”. Timezone should be “time zone”; no, 9.6(24) uses “timezone”. That seems weird, John will check.

Tucker takes an action item to review the AI for any other conflicts with his model of time (where Split does all the work).

Various typos:

- Function Second should return Second_Number, not Minute_Number.
- Function Hour “as appropriate {for} the specified time zone”.
- John asks to change in the summary, “on” to “for”.
- John sees another error in the problem, change “when the load [in] {is} minimal”
- Tucker finds in the AARM Note after function Time_Of, change “A leap second [as]{at} midnight {UTC},.”.
- “2 digit form” should have a hyphen “2-digit form”. There are quite a few of these.

John asks that the Value AARM note be changed “…no [value]{benefit}.”, as the use of value here can be confused with the function being described.

Tucker would prefer to be more specific as to what Value takes: “The string shall be formatted as described for Image, except that leading zeros may be replaced by spaces.” Gary thinks we shouldn’t bother with the flexibility. Just say “shall” instead of “should”, and drop the AARM note. No, “shall” is wrong; that is used as a direction to the compiler, not the user. It should say “Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Time value.”

Approve AI with changes: 9-0-0.

## AI-359/02 Deferring freezing of a generic instantiation

Tucker explains the existing version of the AI. Elaboration of some generic instantiations (only ones that are currently illegal) are deferred until the end of the unit.

John wonders if this problem has occurred in practice (other than the signature package issue). Tucker says it has; they had to do a big redesign because of a problem like this.

Tucker goes on to explain the alternative approach of using a "partial instantiation", so to allow a two-part instantiation.

Ed really does not like messing around with the freezing rules. That code is a nightmare. Tucker thinks that it can be handled by suppressing freezing in the instantiation.

Pascal agrees with Ed that the freezing rules are a nightmare.

Randy explains that the "incomplete instantiation" works well. Tucker says that there is more semantics and syntax with that solution.

Pascal proposes a categorization pragma (close to Preelaborate, perhaps Preelaborate is good enough) and only generics with that pragma would allow this deferring. Then there is no elaboration going on during the instantiation. That would mean that no change can happen by accidentally changing the meaning of the formal types. The specification would be logically expanded at the point of the instantiation. Does this freeze the types that are defined? You would freeze as if it is expanded at the point of the instantiation. Erhard suggests that this rule apply to all instantiations of preelaborated generic packages.

Ed thinks that not freezing would be possible, by adding an extra loop. But this would allow more than it currently does, so that later representation clauses would be OK.

Randy thinks that there would still be some elaboration problems; elaborating the body would allocate the memory for the package body objects. So a call before that elaboration (which would be allowed by this proposal) would die. Pure (not Preelaborate) would work, as it doesn't have that problem.

We take a straw poll on whether to keep the AI alive and continue the discussion. That passes 5-1-3.

Ed would be willing to try some surgery to GNAT to see how hard this is.

Tucker wonders if the relaxation should just be allowed for signature packages. That seems like too much work for a small gain; most of the interesting examples posed so far are using component libraries.

Tucker thinks that calling into an instance that isn't frozen causes trouble. Randy says that he can generate the call, but there better be a runtime check (because you're passing an uninitialized descriptor). Tucker suggests instance freezing — touching something into it would freeze the entire instantiation. Randy says that would work for his implementation.

Pascal does not like that model. He wants it to follow the model of hand-expansion. He thinks that that's all users can understand. Tucker thinks that the code is all together and too hard to separate into tiny pieces.

Randy points out that in their implementation, a composite type's default initialization is a thunk that works like a regular call. If that has not elaborated, you're still dead. Entire instantiation freezing eliminates this problem.

Pascal and Ed think that this is much more work in the freezing rules. They think that expanded freezing falls out in their implementations.

Possibilities:

1) Freeze generic instance on freezing object/type/subprogram declared in instance. (Perhaps apply this to all generics?)

2) Add a pragma that disallows bad things and freeze as though macro-expanded (elaboration of body takes place in the normal place, but does nothing important). (Randy points out that such a pragma can't allow variables in the body.)

3) Existing proposal.

4) Explicit partial instantiation.

The only problem with doing (1) all the time would be that it would be possible for two instantiation's elaboration to occur in a different order. That would be an incompatibility.

We take a straw poll. Is the proposed possibility solution acceptable? (1) 3-3-3; (2) 6-2-1; (3) 1-4-4; (4) 2-7-1

Tucker suggests a composite alternative: add a pragma as in (2), then use the semantics of (1) for freezing.

The instance could have the pragma. Tucker thinks that there should be no checking for the pragma; it would be an assertion. We add additional possibilities:

1.5) pragma Defer_Freezing (on instance).

1.6) pragma All_Private (on generic, with simple checking).

Is possibility (1.5) acceptable? 3-1-5.

Pascal suggests writing up possibility (2) as an alternative, and updating the existing AI with possibility (1.5).

We take a straw vote to decide between (1), (1.5), and (1.6). Should there be a pragma, or should this behavior be automatic? Pragma 6; automatic 2; abstain 2. Should the pragma be on the instance: 6-2-2.

Pascal will write up possibility (2); Tucker will morph the current proposal into a rule based on freezing the instance as soon as you touch something in the instance, with a pragma on the instance (i.e., possibility (1.5)).

## AI-362/02 Some predefined packages should be recategorized

Ed reports that it appears difficult to do these. Randy reports that DDCI and IBM were generally positive.

Pascal wondered if we should allow implementers to make these Pure. We agree that would be a bad idea, because it would limit portability. What about making System Pure? It would avoid the current dependence on whether it's pure or preelaborated, and it would be compatible. Randy reported that DDCI had problems with that, and he thinks he would, too. He should poll implementers again on that.

Fix "preelaboratable" → "preelaborable", etc.

Approve intent: 9-0-1.

## AI-363/02 Eliminate access subtype problems

The summary isn't very professional ☺. Randy points out that he originally inserted it as a joke; he expected Tucker to replace it. Tucker will have to change the summary.

Tucker reads each change and explains it.

In 3.3.1(9), the change is to replace "**aliased**" by "**limited**", and eliminated the explanation.

3.6(11) is deleted (which allows components that are both unconstrained and aliased).

The change to 3.7.1(7) is incompatible with Ada 95. Steve notes that the wording seems wrong, because it doesn't seem to handle nesting well. Tucker claims it does. Eventually we agree with Tucker, but it is confusing. Tucker suggests replacing “a” by “all” in the last sentence. Steve still doesn’t believe the rule works. Tucker says that the “and” doesn’t necessarily mean both at the same time.

Tucker is asked to show an example of the incompatibility.

```
generic
   type Designated (D : Boolean) is ...
package Outer is
   generic
     type Ref is access Designated;
   package Inner is
     ---
   end Inner;
end Outer;

package body Outer is
   package body Inner is
      subtype RefType is Ref (True); -- Illegal.
   end Inner;

   generic
      type Ref2 is access Designated;
   package Inner2 is
      subtype RefType2 is Ref(True); -- Not illegal,
                                     -- checked in instance.
   end Inner2;
end Outer;
```

Pascal asks if we are comfortable with this incompatibility. No one thinks this is common enough to be concerned about.

In 3.10(9), the rules about constrained objects are deleted (this gets rid of privacy breaking issues).

In 3.10.2(26), add text that limits 'Access to components that are constrained by their initial value. Pascal notes that the same rule needs to apply to renamings, in 8.5.1.

In 3.10.2(27), John notes that one of the D’s in the original Standard text is not in italics. The editor should add this to the presentation AI’s notes. But no one can figure out what this rule means. Tucker writes an example:

```
type D (DD : Integer := Y) is record ...
X : aliased D (…);
type A is access all D;
P : A;

P := X’Access; -- Legal? (not with new rule)
P.all := M;
...
```

Why does this need to change? Tucker has trouble remembering at first. It has to do with a type without visible discriminants. The assignment P.all is not going to make any discriminant check (it has no visible discriminants). Some members object that this change is too big of an incompatibility. Tucker suggests that we look at the other changes before drawing conclusions.

In 4.8(6), add text to allow unconstrained in the heap for private types without discriminants. Steve notes that there is a similar problem in generic formal private. Several people think that should be fixed as well. Generic private and private should work the same. But Steve notes that any access type might be passed as an access to a generic formal type, so making the similar change to generic private would require allowing any access type to point at unconstrained in the heap.

The change to 4.8(6) is the performance incompatibility. Some people find it hard to swallow.

Steve wonders if there are any other case in the language where converting a type to be private changes the dynamic semantics of the program. Tucker doesn't know any off hand.

Gary worries about the inconsistency. Tucker argues that we have many rules to fix these sorts of problems in the current standard.

Ed wonders if there is still a way to allocate constrained instances. Removing the defaults would work (which seems best); adding unknown discriminants also would work.

Straw vote: eliminate constrained access subtypes when they cause trouble passes 10-0-0.

Straw vote: unconstrained on heap in limited circumstances is inconclusive 3-1-6.

Gary says that in 3.7.1(7/1) "non-tagged" should be "untagged".

We turn to discussing the effect that these changes have on the other AIs. Tucker explains that AI-295 is about enforcing 3.6(11), if we drop that (as proposed here), we don't need AI-295. AI-275 also isn't necessary if 3.6(11) is dropped, but there is an optimization issue. Without AI-275, a code generator cannot assume that an array object does not have aliased components. But the generic case is uninteresting for optimization; either you have template substitution, in which case the optimizer knows the actual, or you have generic body sharing, which always assumes the worst anyway. So there is no optimization issue with AI-275.

AI-168 (the old corrigendum issue) does matter for optimization, because parameter passing of view conversions can have optimization impacts. Tuck is asked to undo this AI, but with caution

Erhard wonders why we're trying hard to eliminate AI-275. Because it requires authors of reusable components to write two generics if you have a formal array type, one with aliased components and one with unaliased components. For instance, we would need four generics for sorting!

We decide that there shouldn't be legality rules solely for optimization reasons, so kill AI-275, and look at backing out the Corrigendum rules for array conversions.

Should the AI be split? Tucker would prefer to keep it together. He'll mark the text to make it clear which text belongs to specific changes.

Approve intent: 9-0-0.

## AI-364/01 Fixed-point multiply/divide

The fix is to change legality rules to name resolution rules. Tucker notes that square brackets are used two ways in the AI. Also, we should use "either" instead of "both", because it is more compatible with Ada 83, which is the whole point of this AI.

John wonders why the summary says "attempts to alleviate". Because it doesn't affect non-primitive operators, it doesn't provide complete Ada 83 compatibility. John insists that "attempts to alleviate" is redundant; drop one of the words.

Steve wonders if parenthesis should be allowed. Yes, in general we don't want them to make a difference.

Randy complains about the use of "fixed-fixed primitive operation". What is that? Shouldn't it say explicitly what it means? Tucker claims that it is "implied" what it means. Do we really want the RM to "imply" something? Pascal says that there is obviously some confusion about the rule, so it should be clarified.

Gary wonders about the ambiguity in type conversions. Tucker says that does exist, as it does in Ada 83.

Pascal asks about the incompatibility with Ada 95. Tucker says that there cannot be much code that has user-defined fixed-fixed multiplying operators because that doesn't work in Ada 95 (the operators can never be directly visible).

Someone mentions Robert Eachus's example of a fixed-fixed operator in the private part. Tucker says that isn't true - it wouldn't be included, but everyone disagrees, it's primitive and that's what the rule says. Tucker says that's not necessary, perhaps it should say in the same visible part. An operator in the private part would still have the ambiguity — tough.

Ed wonders if literals cause problems. Tucker says they don't, because they don't have other types.

Pascal suggests that this would be a lot of surgery to the resolution code. Tucker explains a possible implementation approach that he says is easy. It fails to convince Ed and Pascal.

Pascal is still worried about the visibility issues. Tucker says that it should be list-of-declarations, not primitive. So that problem should be fixed.

Pascal still thinks it is too much implementation mess for this problem. Tucker says that it is one reason for people sticking with Ada 83. John reports that people running into this get angry. That's not what we want for Ada.

Approve intent: 8-1-1 (Pascal objects as noted above).

## AI-366/01 More liberal rule for Pure units

Erhard asks why generic packages don't allow variables in the body. Tucker explains that the model is that a Pure instantiation cannot look in the body to check whether it is Pure. It could be allowed in the specification, but that doesn't seem very useful. There probably shouldn't be limitations on the formals.

Pascal would be in favor of allowing access-to-subprogram in Pure units. That is included in the wording of the AI.

Pascal has another issue as well, that he discussed privately with Randy, but despite heroic efforts from him and Randy, they can't remember it.

Tucker suggests adding generic formal part to the list of exceptions. Insert that after "generic subprogram".

Move the comma in the wording: "…any variable{,} or named access-to-object type[,] for which…". There is a typo in the summary: "tyes" → "types".

Randy points out that 10.2.1(18) needs to be updated to take access objects into account. (This was a bug in Ada 95.) AI-290 has a revised version of that paragraph.

We decide to discuss AI-290 now.

After a lengthy discussion of AI-290, it was killed. So we need to move the Implementation Permission in AI-290 to this AI, and it should replace 10.2.1(18).

Tucker wonders whether allocators of Storage_Size 0 should be disallowed. Pascal doesn't want representation items to feed back to static semantics. That seems like a sensible policy.

Storage_Size doesn't have to be static. We should say a "static expression with value 0". We can prove that a dynamic expression can't happen (because it would violate some other rule of 10.2.1), but the logic to figure out that result is a pain. It's better to just say static and avoid future questions.

Approve intent: 10-0-0.

Erhard will update the AI.

After the meeting, Pascal reported the issue that he couldn't remember. He believes that imported objects (for which no default initialization takes place) should be excluded from the restrictions of Pure units. That should be considered when updating the AI.

## AI-367/00 Include type declarations for Natural and Positive in package Interfaces

Tucker says that the proposer missed the point: these are hardware mapping types. Hardware doesn't have any subtypes. Moreover, a user-defined package can declare these easily.

Erhard thinks we should at least write up an answer. Pascal will write up the AI for that purpose, based on the mail he sent.

No Action: 10-0-0.

## AI-368/01 Restrictions for obsolescent features

There is only one identifier here (fix summary so it isn't plural).

Steve wonders if having a pragma affect lexical analysis would be hard to implement. It really doesn't change the analysis, some cases trigger an error if the restriction is in place. It does mean handling errors there.

Straw vote on whether to have this restriction: 7-2-0.

Gary voted against, he thinks that Annex J is a bit bogus. He also thinks that Restrictions are not for enforcing style. Ed agrees, but several others disagree. We already have AI-257 with similar restrictions that come from GNAT. So at least one implementation uses restrictions for enforcing style.

We need to decide where this restriction would go. Tucker suggests putting it into chapter 13. The wording should be "There is no use of language features defined in Annex J." Might add a cross-reference to it from J(1).

The restriction shouldn't be partition-wide. The wording should be the same as that used in AI-257.

It would be very hard to implement J.1, because the user could compile the identical code. And that shouldn't be disallowed by this restriction. Make detection of J.1 implementation defined.

Approve intent of the AI: 8-0-2.

Randy will update the AI.

## AI-370/01 Add standard interface for environment variables

There are two proposals here, one from David Wheeler (via SIGAda), and the other from Jean-Pierre Rosen (via AFNOR).

Tucker suggests that perhaps this should be a predefined map. Most disagree; this should be kept simple if possible.

Jean-Pierre's proposal includes path functions. These don't seem to be special enough to include here.

Tucker suggests a Count of objects, a Get_Key(Position), Get(Key, Value), Set(Key, Value), Exist(Key). Ed says that David Wheeler's package is close to this. But the names in that proposal are terrible, as is the description of the subprograms (which is described in terms of the C library).

The AI should be updated to make a reasonable proposal. Pascal is drafted to do this.

Keep it alive: 8-0-2.

## AI-371/01 New hierarchy for OS-Dependent services

There is no interest in another hierarchy, Ada is fine for things that depend on the target operating system.

There is also no interest in OS_Name and OS_Version; they don't seem sufficiently useful given that they return implementation-defined information. Having them doesn't add anything to portability.

Randy suggests that the Processes package would be useful. Pascal says this looks like VMS. Ed says that it is similar to GNAT's OS_Lib. He thinks it makes sense to standardize it, it's commonly-used functionality.

Tucker wonders if it is important enough, and it isn't useful enough by itself. There are other communication mechanisms that are commonly used that are not covered here, and even in the restricted context of starting another process, the proposal doesn't address issues like I/O redirection, passing environment variables or credentials, selecting a shell, etc.

Erhard would like a package like Processes. Don't most implementations have a package like that? GNAT and Janus/Ada do. IBM Rational does not.

No Action: 6-1-3.

### AI-372/01 Restrictions for default stream attributes of elementary types

This proposal came from AFNOR at the deadline.

There is only one identifier here (fix summary so it isn't plural).

Steve wonders if you can make this check for class-wide operations. That seems problematical. Randy and Ed wonder if this shouldn't be checked when items are declared. Tucker points out that doesn't handle the ones in Standard.

Tucker claims check-on-use is easy. Randy points out that shared code generics would not know; the implementation is similar to class-wide calls. Tucker claims that raising Program_Error would be good enough in this case.

Tucker thinks this solves the wrong problem; he'd prefer more control over default sizes of elementary. Randy points out that we already did that in AI-270 (which is already approved). This is needed since we can't control the sizes of types in Standard. Pascal suggests that this whole issue would be better handled with an ASIS tool.

No Action: 10-0-0.

### AI-373/00 Undefined discriminants caused by loose order of init requirements

The problem can occur within components, as well as within discriminants. A solution only covering discriminants is insufficient. Gary had suggested using rules similar to the rules for finalization of types with per-object constraints (7.6(12)).

Tucker would like to just say that the discriminants are handled first. No, the idea is that everything that doesn't depend on the discriminants is done first.

Steve Baird will update the AI.

### AI-374/01 Assertions_Only pragma

Tucker updated the AI, splitting it from the pragma Assert one. He moves No Action. Since these are pragmas, anyone could do them, and there is not any implementation of a similar pragma now.

No Action: 8-0-0.

### AI-375/01 Type and package invariants

Tucker updated the AI, splitting it out of preconditions. He moves No Action. As with preconditions, he doesn't think that we can finish it in time to meet the schedule.

No Action: 8-0-0.

## ***Detailed Review of Regular AIs***

### AI-291/02 By-reference types and the recommended level of support for representation clauses

Tucker would prefer that the wording be consistent with other cases of recommended level of support in the Standard. “An implementation need not support <blah> if …”

Should “confirming” be defined? Yes, because it is used several times, and AI-51 and 109 probably will use it as well. Move that paragraph to the end of static semantics.

Tucker worries that specifying a representation clause might change the semantics. That is a bad idea; the RM should probably be fixed when that happens. For instance, AI-247 deleted 13.3(26) for this reason. 13.3(50) also has a dependency on “specified”, so it should be fixed too. We discuss that for a long time, getting far off topic. (This was the last AI discussed at the meeting.)

The NOTE should be an AARM Note; it’s aimed at implementers, not users.

Paragraph 2 of the replacement for 13.1(24) means that all aliased objects have the same size and alignment. It would be better to say that clearly. Tucker says that size clauses don’t change the layout of component types anyway (at least that’s not required); so this is really needed only for elementary type.

Is volatile relevant? Pass by copy is required for elementary. The alignment has to be correct for volatile objects, which might be passed by reference. (That only applies to composite.) But such things are already by-reference, so they are already covered. Steve is asked to check C.6 to verify what exactly is covered by the fact that volatile types are by-reference.

Tucker suggests 13.3(72) should say “...need only support confirming…”. Then he leaves.

Approve intent: 5-0-2.

### AI-360/03 Types that need finalization

Correct summary: “A {number} of language-defined…”

Change the second bullet of the wording to: “it has a component that needs finalization”.

Change the third bullet of the wording to: “it is a limited type that has an access discriminant whose designated type…”

Do the random number generators need finalization, because they are often implemented with a controlled type (so as to avoid the Rosen trick)? “The type Generator needs finalization.” Both Generator types should have this added.

Add (see 7.6) after “needs finalization” in all cases.

Tucker would like an AARM Note that just because a type “needs finalization” does not require it to be implemented with a controlled type.

Approve AI with changes: 8-0-0.

### AI-369/00 Completions and renaming

Inline works through renaming, so Import ought to as well. But there is an easy workaround (use a non-overloaded name, and rename to that). No one cares enough about the issue to take on the AI.

No Action: 8-0-0.