Minutes of 13th ARG Meeting

18-20 May 2001

Leuven, Belgium

Attendees: John Barnes, Randy Brukardt, Kiyoshi Ishihata, Pascal Leroy, Stephen Michell, Erhard Ploedereder, Tucker Taft, Joyce Tokar.

Observers: Ben Brosgol (of ACT, USA) (Friday only), Brian Dobbing (of Praxis Critical Systems, UK) (Friday only), Steve Baird (of Rational, USA)

Meeting Summary

The meeting convened on 18 May 2001 at 14:00 hours and adjourned at 15:00 hours on 20 May 2001. The meeting was held at the Maria-Theresia College in a room provided by Ada-Europe on Friday and at the Holiday Inn Garden Court on Saturday and Sunday, both in Leuven Belgium.

The meeting covered the entire agenda, as well as brainstorming possible solutions to problems raised by several amendment AIs. The first two days of the meeting were spent on amendments. The third day was spent on normal AIs.

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

Next Meeting

The next meeting is set for 5-7 October 2001 in Minneapolis, Minnesota, in conjunction with the next SigAda Conference. The ARG meeting will begin in the early afternoon of 5 October, after the WG9 meeting has adjourned.

Some discussion ensued about the following meeting. Waiting until Ada-Europe (in June) is too long between meetings; the ARG has typically met three times per year. A meeting at the Embedded Systems Conference in San Francisco (in March) was suggested.

WG9 Actions

WG9 has approved a new charter for ARG.

Charter of the Ada Rapporteur Group

The Ada Rapporteur Group (ARG) is a subgroup of ISO/IEC JTC1/SC22/WG9, the JTC1 Working Group for Ada. The ARG has the following duties:

- *Serve as an advisory group for the project editors of ISO/IEC 8652 and ISO/IEC 18009.*
- *Support the SC22 defect and interpretation process by drafting publicly available responses to Defect Reports on ISO/IEC 8652 and ISO/IEC 18009.*
- *Draft text for proposed clarifications, corrections and changes to those two standards and others as assigned by WG9.*
- *Recommend strategies for extensions of the Ada language and libraries and prescription of conformity and optionality via the use of corrigenda, amendments, secondary standards, technical reports and informative materials.*
- *Coordinate with other organizations to promote uniform implementation of the Ada standard and appropriate usage of Ada in other standards.*

Language proposals originating in other Rapporteur Groups of WG9 will be referred to the ARG for disposition in the same manner as suggestions originating outside of WG9.

WG9 also has approved a New Work Item for amendments to Ada 95. We couldn't get secondary standards included in this, unfortunately.

New Chair

Erhard announces his intention to resign as chair. He will continue as a member of the ARG. He nominates Pascal Leroy to succeed him.

Tucker moves to vote. John seconds.

Pascal is elected by acclamation. (Note that technically the chair is selected by WG9, so this is really a recommendation to WG9).

Old Action Items

The old action items were reviewed. One AI was given very low priority. Following is a list of items completed (other items remain open):

John Barnes:

- AI-241

Randy Brukardt:

- AI-85
- AI-174
- Post existing test objectives and assignment to the Grab Bag on the ARG web site;
- Publish proposed test objectives to ARG
- AI-218
- Standard library enhancement for directories (AI-248)
- AI-227
- AI-235
- AI-236
- AI-237

Pascal Leroy:

- Exception hierarchy (AI-264)
- Raise exception with objects (AI-264)

Steve Michell:

- Tagged protected types (AI-250)

Tucker Taft:

- Object'operator (AI-251)
- Interfaces and Multiple Inheritance (AI-252)

Joyce Tokar:

- Ravenscar Profile (AI-249, now additionally AI-265 and AI-266)

Mike Yoder:

- AI-185

All:

- Review proposed test objectives and vote on these objectives

AI-186 (was assigned to Pascal Leroy) was downgraded to very low priority.

New Action Items

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

Steve Baird:

- AI-263

John Barnes:

- Downward closures

Randy Brukardt:

- AI-218
- AI-224
- AI-229
- AI-237
- AI-248
- AI-257
- AI-258
- AI-260
- AI-262
- AI-268
- Split second issue from AI-263 into a separate AI.
- Object_Size attribute

Gary Dismukes:

- AI-158
- AI-196

Bob Duff:

- Be the test creator of last resort
- Fix limited types a bit to permit aggregates and initialization by function call [jointly with Tuck]

Mike Kamrad:

- Amendment on Assert pragma
- Various items to be standardized [jointly with Mike Yoder]
  - Discard_name & ‘image
  - External_tag
  - Storage_IO of tagged types

    - Array indexed by holey enumeration
    - Static elaboration
    - GNAT attributes and pragmas
- CPU time (separate from real-time) [jointly with Joyce]

Pascal Leroy:

- AI-195
- AI-230 [jointly with Tuck]
- AI-255
- AI-264

Steve Michell:

- AI-148
- AI-250

Tucker Taft:

- AI-133
- AI-162
- AI-188
- AI-214
- AI-216
- AI-217 (on hold pending abstract package proposal)
- AI-230 [jointly with Pascal]
- AI-231
- AI-251
- AI-252
- More stream representation control (split from AI-195, now AI-270)
- User defined assignment/subtype conversion
- Physical units (length/dimensional analysis), whereby square inch units are generated by result of multiplying inches; subtypes with special attributes would be used to provide the specifications of these dimensions, reducing the introductions of lots of extra operators.
- Fix limited types a bit to permit aggregates and initialization by function call [jointly with Bob]
- Abstract package proposal
- Task group proposal

Joyce Tokar:

- AI-249
- AI-265
- AI-266
- CPU time (separate from real-time) [jointly with Mike Kamrad]

Mike Yoder:

- Various items to be standardized [jointly with Mike Kamrad]
    - Discard_name & ‘image

    - External_tag
    - Storage_IO of tagged types
    - Array indexed by holey enumeration
    - Static elaboration
    - GNAT attributes and pragmas

All:

- Create tests for assigned AIs [still outstanding]
- Recommend new members

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.

Detailed Review of Amendment AIs

AI-217: Handling mutually dependent type definitions that cross packages

Tucker would like to abandon the **with type** proposal. He says it is misguided, because you have to posit the existence of “ghost” or “incomplete” packages to describe it. In other words, a compiler will need to create a hidden package specification, and populate it with the right stuff. Then it is possible to see a lot of these ghost packages, which provide different views of the same package, and the compiler must figure out that they all correspond.

So Tucker says: if we have to have ghost packages, let’s make them explicit. He shows an example of his idea, which he names a “package abstract” (note the use of “abstract” as a noun here):

```
[with abstract Q;]
package abstract Q.P is
    pragma Preelaborate (Q.P);
    type T1;
    type T2 is abstract; -- See AI-251
    type T3 is tagged;
    type T4 is access constant T1'Class;
    pragma Convention (C, T4);
end Q.P;
```

In addition to the above constructs, nested packages should be allowed in the abstract, too.

The elaboration of a package abstract has no effect. This implies that all expressions have to be static.

Steve B. and Pascal immediately raise an objection: what if elaboration code is generated to create a storage pool for an access type? Tucker says that no code is generated here, it is generated on the full specification. Therefore, it is necessary to prevent the use of allocators on types imported from an abstract.

Steve B. wonders if this was restricted to just incomplete types, would that be enough to solve the problem? Discussion turned to the restrictions on the contents of an abstract. Steve B. suggests incomplete and abstract only. This includes interfaces and “tagged incomplete” as in the **with type** proposal. We seem to be back to a discussion we had on the **with type** proposal, where **with type … is access** was discarded because of difficulties with representation and operational items.

Tucker believes that the solution is to require enough information to be given in the abstract to determine representation issues. Representation items for an access type are allowed in an abstract, since they make it possible for clients to generate the appropriate code (representation items pertaining to the storage pool would probably have to be disallowed, though, since the collection is only created later). The full specification cannot contradict the information provided in the abstract. Tucker suggests that representation clauses should be repeated (in a conforming manner) in the full specification, because people expect to read the specification to get all of the information.

Randy wonders how you can make “enough information” portable. Tucker replies that you have to given everything that is necessary. Randy is not satisfied by the answer.

Pascal suggests that the abstract is essentially a partial freezing point; incompatible representation clauses are rejected on the compilation of the real specification.

The abstract specification is essentially the third (or zeroth) part of a package.

Steve B. asks if the completion (the full specification) could be an instantiation or library-level renaming. Tucker thinks that an instantiation would be OK, but a renaming would be bad.

Someone asks if the parent dependency of a child abstract needs to be explicit. Tucker proposes the following: if there is an explicit “**with abstract** Q;” on unit P.Q, the parent dependence is abstract; otherwise the real Q is visible.

In terms of library management, the situation is somewhat similar to subprogram specifications and bodies. In particular, the specification of a package has a semantic dependency on its abstract if there is one.

Erhard wonders if this is the start of a package interface? No, it is more like an incomplete package.

Discussion suggests writing a new AI to fully flesh this out, give it a new number. John suggests to use the same example to make the AIs comparable. Tucker will create the AI, sooner rather than later.

AI-218: Accidental overloading when overriding

Only minor editorial changes were needed. The last paragraph of the wording must be corrected as follows:

> Explicit_Overriding applies if and only if it applies to {the} generic unit. At a [declaration]{place} where a …

and the following typo must be fixed:

> … an overrid[d]ing declaration.

Steve M. is concerned that configuration pragmas are partition-wide. Erhard replies that they are not defined that way in the RM. Moreover, it is very useful semantics to localize the effects of the pragma, as library code might be part of the partition, but not adhere to the use of the Overrides pragma.

Approved with editorial changes 7-0-1 (Steve M. abstains).

AI-230: Implicit conversions between access types

Tucker proposes to make anonymous access types first-class citizens, and allow them in object and component declarations (in record and protected types). Having finished the regular agenda, the ARG held a brainstorming session to explore this idea further.

Tucker thinks that if we do this for anonymous access types, it would be sensible to do the same for anonymous array types. In fact anonymous array types were allowed as record components in an early version of Ada (Ada 80). The point is made that the reason why this was removed was that nesting of types is problematic. Tucker suggests that the “inner” (or component) type must be hoisted so as to be declared before the outer type. (This will cause some problems for pure one-pass compilers).

For various reasons, we don't want the component subtype or designated subtype to have constraints. So we would want to use subtype_mark, not subtype_indication, in the new productions:

```
array (...) of subtype_mark;
access [all | constant] subtype_mark;
```

(Of course, an anonymous array object currently has a subtype_indication as its component subtype, and that case would have to remain legal, so the syntax would need some surgery.)

Erhard believes that it would be useful to declare a reference type as:

```
subtype S is access T;
```

It is noted that this would mean that we have named anonymous access types!

To accomplish this, Erhard suggests that anonymous types be allowed in subtype_indications:

```
subtype_indication ::= ... |
      access [all | constant] subtype_mark;
```

Randy reads the list of uses of subtype_indication; this change causes problems with allocators, derived_type_definitions and access_type_definitions (we want to disallow things like **new access** T —in its two incarnations— and **access access** T). So we have to use a new non-terminal to disallow anonymous access types in these contexts: the syntax definitely needs some surgery. An additional restriction would be to disallow deriving from an anonymous access type (assuming that these work in subtype declarations).

At this point the group discussed the case of anonymous array types, which Tucker would want to allow. Anonymous array types would only be useful if they came with implicit conversions, but the bounds may be different, so some form of static compatibility would have to be required. This seems to make the proposal significantly more complicated, and chances are that this could kill the entire proposal. It is suggested that anonymous access types be dealt with first, because they seem more important, and that anonymous array types be handled by a separate AI, if they are deemed useful.

Questions come fast and furious from the group: Do anonymous access types have "=" and **null**? Tucker says yes, but he believes that we should not allow users to redefine "=". His first proposal is that there is an implicit declaration of "=" (**access** T, **access** T) for each anonymous access type **access** T. But this doesn't work, because it would create lots of ambiguities. Moreover, Erhard does not like the notion of "=" being defined at the location of the type, because it makes it hard to find these operators in order to use them.

Tucker's second proposal is to either have one "=" in Standard (with *universal_access* parameters), or none. But putting one in Standard would be very ambiguous. We would need a preference rule, but this would create the risk of a Beaujolais effect. Tucker concludes that we shouldn't have an "=" operator. Steve B. points out that we have to have predefined equality, or records containing these could not be compared. Tucker comments that we also need a null test if these are allowed to be **null**. Pascal comments that we also need that for parameters if we allow null anonymous parameters.

Tucker's third proposal is to have one magic "=" operator in Standard, with two parameters of type *anonymous_only*:

```
function "=" (Left, Right : anonymous_only) return Boolean;
```

This operator would only be considered by the name resolution rules if at least one of the actual parameters is of an anonymous access type.

Pascal and Steve B. both mention that this allows comparing **access** Integer and **access** Float. So Tucker adds yet another magic legality rule: "and by the way, the designated subtypes of the actuals must statically match".

Steve B.: how about we declare such a thing for every type T:

```
function "=" (Left, Right : access T) return Boolean;
```

We return to the one in Standard. This is always visible; so it would hide other operators (when referenced by a **use** clause).

Discussion then turns to accessibility levels. Tucker suggests the following rules:

- Discriminants - object accessibility (also can have a subtype declaration here, which determines the level)
- Parameters - dynamic accessibility
- Components - level of record/protected type
- Objects - level of object
- Subtype declarations - level of subtype declaration

We then took a lunch break. Erhard worked on the proposals during lunch. When we returned, every writing surface was filled with examples and questions. He wants to understand the rules regarding all cases of conversions to and from access-to-specific and access-to-class-wide types, and rules for dispatching and name resolution.

Example 1:

```
type T1 is access T;
type T2 is access T;
X : T1 := …;
Y : T2 := …;
Z : access T;
Z := X;
Y := Z; -- Storage pool issue: Y now points in the storage pool of T1
```

This is illegal. Implicit conversion from named access type to anonymous access type is fine, but there is no implicit conversion from anonymous access type to named access type. Hence there must be an explicit conversion, which is illegal because T2 is not a general access type. In other words, the existing rules do the right thing here.

Example 2:

Given the types:

```
type T1 is new T with ...
type T1Acc is access all T1'Class;
type Tacc is access all T'Class;
type TPtr is access all T;
```

Which of the following conversions are allowed, implicitly or explicitly?

```
access T → access T'Class –- implicit
access T → Tacc –- explicit
access T → TPtr –- explicit
access T'Class → access T –- explicit, possibly via P.all'Access
TPtr → access T'Class –- implicit
access T'Class → TPtr -- explicit
access T1'Class → access T'Class -- implicit
access T1'Class → Tacc -- explicit
Tacc → access T1'Class –- explicit, possibly via P.all'Access
access T'Class → access T1'Class –- explicit, possibly via P.all'Access
T1Acc → access T'Class -- implicit
```

We can tell by using the "covers" rules, and the fact that conversions to named types are always explicit. Note that these rules are not new; they exist in Ada 95.

At this point someone comments that "named anonymous access types" provide a nice alternative to P.**all**'Access for converting a value of a named access type to an anonymous access type.

Example 3:

Does **access** T1'Class match **access** T for dispatching primitive operations. (No.)

Does (**access** T1'Class).**all** match T for dispatching primitive operations. (No.)

It is pointed out that this problem is solved by the Obj.Proc notation for calls (AI-252), so it does not matter.

Erhard is eventually satisfied, assuming that named anonymous access types exist.

At this point, Steve B. takes over the marker, and puts up the following example, which illustrates a difficulty with the accessibility rules as outlined above:

```
X :access T;

procedure P is
      Y : access T := …
begin
      X := Y; -- Accessibility check?
end;
```

Tucker responds that this is illegal because of static checking of accessibility checks. This is judged to be a problem because it might occur in a record type that came from a generic instantiated in a frame.

Runtime checks are considered. But runtime checks cannot be done by simply comparing the static levels of entities. (Simple comparisions don't work if you have general assignability, just as they don't work for access parameters in Ada 95.)

It is noted that runtime checks aren't necessary for objects declared at the library level.

The idea is that the implementation would keep the level of what is stored in an anonymous access object/component, and never allow a shorter lifetime assignment to a longer one.

Should functions be able to return these? John comments that this would potentially allow returning of references to terminated tasks outside of their masters, a problem that we spent a lot of effort to eliminate in Ada 95. But Tucker shows a use for functions returning these:

```
F (X'Access).all := …; -- Where F is a component selector for X.
```

The implementation would track the level of the parameter, and the level at which it is returned.

Most agree that static checks will be too restrictive. Accessibility looks like a real can of worms.

Tuck floats a trial balloon that the accessibility level is the same as that of the designated type. This is a lead balloon: you couldn't assign an access to a local variable into a local object of one of these types. However, it is still useful to notice that if the accessibility level of an anonymous access is the same as that of designated type, then no accessibility check is needed.

There is general agreement that runtime checks are necessary. Randy objects to the runtime cost, especially because of the uncertainty how the checks are performed. It is pointed out that many of these checks and costs can be optimized away. Randy is unconvinced, it seems very heavy to him.

Erhard weighs in saying that runtime checks aren't necessarily expensive, because the anonymous type could be declared globally and hence no checks necessary. But we can't require global declarations, as we want to allow 'Access on locals. In fact, this seems like a useful capability provided by "named anonymous access types": by declaring a subtype at a high level, you can raise the accessibility level of an anonymous access type.

Steve B. worries about distributed overhead for doing this checking, because the existing scheme of runtime accessibility checking doesn't work. Tucker is still convinced that it does work. No firm conclusions are reached.

Action item: Tucker should rework AI-230 with all this discussion in mind. Anonymous arrays should be handled separately.

AI-248: Directory Operations

Randy noted that he didn't have time to research LDAP for inclusion in this AI. Tucker says that LDAP is too heavy for this AI. Randy notes that the minutes of the last meeting say that Tucker asked him to do that research, so he will not bother with fulfilling that request.

It was noted that the package seems to be too high in the hierarchy of predefined packages. Randy notes that this package works with all sorts of files, so it would be inappropriate for it to be a child of any specific IO package, and making it a child of IO_Exceptions also seems inappropriate. If Ada 95 had defined a package IO that was the parent of all of the IO packages, then this package could go there, but of course Ada 95 does no such thing. Moreover, we could cause conflicts by defining grandchildren of Ada, but not for children of Ada (since it is illegal to compile children of Ada, and implementors should not define their own, while there are no such restrictions for grandchildren of Ada).

Change the name of Get_Current_Directory to Current_Directory.

Steve M. suggests that Remove_Directory ought to say that non-empty directories are not deleted. After some discussion, it is suggested to add Create_Tree (which makes all directories needed in the name) and Remove_Tree (delete everything in the named tree, including directories) to the package. Steve's suggestion is adopted.

It is also suggested to add Delete (by name [string]) to the package. Rename (Old_Name, New_Name) should also be added. This should be described as changing the name of the file from Old_Name to New_Name. There should be an implementation requirement for this to work at the same level in the same directory (others may raise Use_Error).

The Name routine for a directory entry should be split into a Simple_Name and Full_Name routine.

Pascal suggests that there should be some way to get the parent directory.

First Erhard, and then Steve M. express confusion about how the directory searching iterator works. Randy notes that if Ada experts can't figure out the POSIX iterator mechanism, what chance do users have? It is suggested to add a limited (type) handle searching mechanism, rather than the current iterator scheme. The suggestion is made that the directory searching stuff be put into a separate child unit to avoid clutter.

Pascal comments that he doesn’t like all these operations taking file names, and that he would prefer a full-fledged Directory_Entry abstraction, much like Text_IO has a File_Type abstraction. There would be an operation to resolve a name to obtain a Directory_Entry, and various operations on Directory_Entries. The group appears unconvinced.

A request for file name composition/decomposition functions was made by Tucker. Randy notes that these are very tough to do right. The group would like to see these.

Steve M. would like to see a way to create links. Others note that the links on Windows or Mac aren't the same as the links on Unix, so a common definition would be hard.

Randy will revise the AI.

On Sunday, Randy brings up a recent conformity assessment problem, which is somewhat related to this AI.

A.8.2(22) says “If an external environment allows alternative specifications of the name (for example, abbreviations), the string returned by the function should correspond to a full specification of the name.”

The ACATS has always tested this as if it were a requirement. However, some implementations have not been following this requirement (and this fact was overlooked by the testers). Should this “should” be a “shall” (a requirement) or shall it remain a “should” (an Implementation Advice)?

There is definitely a serious overhead (it is necessary to call getcwd() on every Open and Create), and there are implementations which have strayed. The best solution is to give users an appropriate directories package, and eliminate this requirement altogether.

Randy says that the immediate problem is three ACATS tests (from ACVC 1.11) which require this behavior. With the above solution, the best course of action is to withdraw these tests (they have no other value).

The general agreement is that a good directories package makes this requirement unnecessary. We should plan to remove this requirement when a directories package is standardized. No one supported making this a “shall”.

Randy will integrate this (and associated operations, such as a Simple_Name → Full_Name routine) into the directories package.

AI-249: Ravenscar Profile for High_Integrity Systems

Joyce presents an overview of the Ravenscar AI. It consists of a pragma Runtime_Profile and new identifiers for pragma Restrictions. (The latter is misdescribed in the AI writeup.)

Tucker suggests using Simple_Barrier_Expressions rather than Simple_Barrier_Variables. Brian points out that the intent was indeed to restrict barriers to variables in the private part of a protected object; even constants are disallowed. John points out that the minutes of the previous meeting suggest calling this Simple_Barriers. Erhard remarks that there is no problem allowing static expressions in barriers (it can't make analysis harder or performance worse). There is general agreement that the restriction should be called Simple_Barriers and that static expressions should be allowed.

Tucker asks if profiles can be additive? (It seems possible to add restrictions.) He means using more than one profile in the same partition. Joyce wonders how conflicts would be handled. Brian notes that there is nothing in the proposal restricting the use of multiple profiles in a partition. The general agreement is that multiple profiles in the same partition are fine, and that their restrictions are additive.

Tucker asks about the purpose of the argument list in pragma Runtime_Profile. Brian answers that it is for future use; it is not used in the current proposal.

Erhard wonders why the pragma is called pragma Runtime_Profile; he thinks it really is an application profile.

Brian thinks that Profile is a fine name for the pragma. Steve M. remarks that Runtime_Profile is what it is. He believes we'll have to describe them somewhere in any case. No consensus is obvious, so a vote is taken. Runtime_Profile gets 2 votes; Profile gets 5 votes, 1 member abstains.

Erhard wonders why there is a requirement for some bounded errors to be detected. Brian explains that Ravenscar implementations are intended to use Ceiling Priorities without locks, but this only works if the implementation prevents potentially blocking operations in a protected action. Thus these bounded errors must be detected. He notes that this rule does not apply outside of Ada (foreign code might suspend, but we can't require checks on that).

Erhard suggests that this rule should say that any “potentially blocking” operations is detected and to put the enumeration of cases into an annotation. Otherwise this rule is fragile: other changes to the standard might break this. The reason why the AI only enumerates three cases is that the other cases are either (1) prevented by the Ravenscar restrictions or (2) outside the control of Ada.

Brian replies that this change seems like a good idea. But the wording must take care not to include stuff outside of Ada's control. Then he changes his mind. He is directed by the group to change his mind again to use Erhard's suggestion.

Pascal remarks that the new restrictions identifiers must be defined separately from the Ravenscar profile identifier. The presence of a **pragma** Profile (Ravenscar) has the effect of enforcing these new restrictions. This is generally agreed.

The group studied the names of the restrictions identifiers. They seem inconsistent. However, the existing Ada 95 identifiers aren't very consistent, either. Several name changes were suggested, but then rescinded when examples of the original style were found in the RM.

The following changes were made, however:

- Max_Entry_Queue_Depth → Max_Entry_Queue_Length
- No_Requeue → No_Requeue_Statements
- No_Task_Attributes → No_Task_Attribute_Package (Straw vote: 5-3-2)

While considering restrictions identifiers, a discussion of Task_Ids broke out. The question was raised whether these (and the package Ada.Task_Identification) are allowed in the Ravenscar profile. The proposal does not restrict the use of this package, and it was indicated that this was intended (unavailable operations raise Tasking_Error).

Joyce and Brian will update AI-249.

AI-250: Protected Types, Extensible, Tagged, Abstract

Immediately, the question is raised: "Do we need this? Are there compelling examples?"

Steve M. tries to give an example, involving difficulties while interfacing to CORBA in a multitasking environment. The user has to use a semaphore lock. Joyce clarifies the example a bit: there is a need to dispatch to the appropriate operation based on the CORBA ID.

Tucker opines that if we're unsure about the importance, we should try to make the proposal as simple as possible, to minimize the changes to the language and the implementations, and to make it more likely that the proposal will be accepted.

He doesn't like the notion of exposing the barriers in the specifications. He thinks that strengthening the barriers is not sufficient, because the condition might have changed when you pass the buck to the parent type or do a requeue, i.e., there is a race condition there. He believes it's sufficient to (1) take the entry as is or (2) completely override it.

Steve M. comments that if you need to extend the type, you need to be able to change the synchronization semantics. Generally, this means barrier strengthening.

The group discussed the example from the AI (Resource_Controller). Here is a possible use:

```
with Rsc_Controller; -- Package where resource controllers live
package PP is
      …
      PT : Rsc_Controller.Simple_Resource_Controller'Class;
      …
      PT.E;
end PP;
```

Tucker suggests eliminating support for handing off to the parent to simplify the proposal. Steve M. notes that this is a common part of the object-oriented paradigm. Tucker claims that there are serious problems with doing that. He would like to see an example where it is important.

Steve M. believes that you wouldn't be able to use entries in extensions if you have multiple levels of extension.

Tucker thinks that the current **and when** proposal doesn't handle the necessary precondition (because the protected type components are changed.) He suggests that a private entry in the parent might be a useful way to provide this functionality. Steve M. concedes that a private entry or procedure might provide most of what is needed. Tucker makes the bold claim that partial reuse without pre-planning never really works in practice. Unintentional reuse doesn't happen.

Steve M. notes that Tucker's proposal requires to totally replace entries in extensions. (These entries allow subprogram calls & requeue to the parent). He argues that refining the barriers is useful because it is somewhat equivalent to preconditions. Tucker says that precondition and synchronization barriers are totally different things. Erhard wonders how the implementation is supposed to work anyway, since with Steve M.'s proposal you may have queued calls with different barriers.

A summary of the two proposals:

- Steve's: Can call parent, when doing so, barrier must be true. The call is essentially a procedure call.
- Tuck's: Can't call parent (essentially, always dispatch). Use private entries to get around the restriction.

Both proposals use a single queue.

Tucker points out a hole in Steve's proposal: What if the parent does a requeue? Then we have a requeue of a procedure call.

Tucker continues and points out a hole in his own proposal: the need to get visibility to the private part of protected types in an extension, which is problematic because protected objects are not library units and don't have children. After some discussion, the suggested rule is that you can get visibility to the private part of your parent only if you have visibility to the enclosing unit's private part.

Someone suggests that these extensible protected types really ought to have their own separate generic formal type. And we would need to define how they interact with generic formal types (e.g., can they be passed to a generic formal tagged limited private type?).

Erhard wonders where the requirement for this feature comes from. It certainly isn't in Java or C++. Java provides for extension of (partially) synchronized classes, but barriers need to be modeled by wait/notify, i.e., condition variables, whose "barrier" is inlined code within the method implementation. Tucker describes the Java model as somewhat brain-damaged.

Tucker notes that this is good more for interface inheritance, rather than implementation inheritance. It also can be used for lock inheritance. Pascal thinks that we should keep lock inheritance separate from interface inheritance. Lock inheritance appears to be a useful capability, which is hard to achieve by other means.

Joyce echoes others in asking, "What is the need for this?" She suggests that we should let it sit and let the community find a need before proceeding. Someone notes that it is a chicken-and-egg problem; it's unlikely that anyone will find a need for it until an implementation is available. Steve M. comments that there is a reference implementation that's getting started. Tuck argues that it's still useful to have a proposal written down so that (1) we can mature it as needed and (2) it serves as a reference for would-be implementers.

The AI is returned to Steve M. for simplification. Randy notes that the AI's title should be something other than a list of keywords.

AI-251: Tagged Types, Abstract Interface, Multiple Inheritance

Tucker starts by saying that he would like to call these "fully abstract types". He then describes an example of their use: (assume these are in appropriate package specifications)

```
type Stack is abstract; -- A fully abstract type.
procedure Append (S : in out Stack; E : Elem) is abstract;
function Length (S : Stack) return Natural is abstract;
procedure Remove_Last (S : in out Stack; E : out Element) is abstract;
```

--

```
      type Queue is abstract; -- Another fully abstract type.
      function Length (Q : Queue) return Natural is abstract;
      procedure Append (Q : in out Queue; E : Elem) is abstract;
      procedure Remove_First (Q : in out Queue; E : out Element) is abstract;

--

      type Deque is abstract new Queue and Stack; -- A fully abstract type
                                                  -- which inherits from both
                                                ..-- the Queue and Stack
                                                  -- interfaces.
      procedure Prepend (Q : in out Deque; E : Elem) is abstract;
      procedure Remove_Last (Q : in out Deque; E : out Elem) is abstract;

--

      type My_Deque is new Deque with private;
private
      type My_Deque is new Blob and Deque with record
                 …
            end record;
      -- Conventional type implementing an interface.
```

John asks Tucker to confirm that fully abstract types can't have concrete operations; regular ones (like My_Deque) can have concrete operations. This is correct.

Tucker continues describing the rules. There can be at most one conventional type in an extension, the one occurring after the word **new**. That's because we don't do full multiple inheritance. A fully abstract type has no components or code.

Erhard notes that the proposal requires non-contiguous dispatch tables.

Tucker describes an implementation model: a descriptor containing two pointers, one at the data, including a normal tag, and the other pointing at a "view dispatch table", a table specific to a particular view. This implementation is only necessary for class-wide access types and parameters of a fully abstract type. This implementation allows dispatching in constant time. Other implementations (generally more expensive) are possible.

A view conversion upwards in the derivation chain requires adjusting the run-time descriptor; a downward view conversion additionally requires the usual run-time check on class containment of the actual object.

The derivation graph for the example above is as follows (italics marks types that are fully abstract):

```
    Root_Type
    |       |
 Stack     Queue
      |     |
      Deque
        |
Blob    |
   |    |
  My_Deque
```

The technical problem is on converting Deque to Stack or Queue, because the dispatch vectors are not "subset congruent" as they do in single inheritance, that is, the methods don't occupy the same slot positions.

The syntax is similar to that used for Ada 95 so that existing code can be converted to use fully abstract types without having to change the syntax. The only change that is needed is to make the root of the type hierarchy fully abstract.

Steve M. asks if this could require multiple indirections in the worst case. It is noted that Ada always allows bad implementations. Tucker explains that Java does this at runtime, but only one level of indirection arises. In Java, the calls have a search, but the conversions don't have a runtime cost. As usual, you can move the bump under various rugs, but you can't get rid of it.

When questioned, Tucker thinks that fully abstract types should not match generic formal abstract limited private types, or act as a completion for an incomplete-type-declared-in-a-private-part-and-completed-in-the-body. That’s because of the 2-word-pointer implementation: as soon as we have fat pointers, we have potential generic sharing problems.

Erhard comments that the terminology is a concern. From the perspective of selling the idea, calling these “abstract types” is confusing. It is best to treat these as a separate world. Moreover, the lack of a special syntax for declaring interfaces is an issue.

Randy weighs in: if you look at this as a small change, then it might as well be a small difference in syntax and terminology. But if it is a large difference, then we should make it appear different.

Pascal thinks that this argument sounds like tagged vs. class from Ada 9x.

Someone suggests that we should just change the terminology (not the syntax). Perhaps, just call them “interfaces” vs. “fully abstract”. John says that he doesn't like adjectives.

Someone notes that Ada has a tradition of names (terminology) different than syntax: Float is spelled “digits”. This triggers a general attempt to find more examples: fixed point is spelled “delta”, class is spelled “tagged”, and so on.

Erhard drops his objection to the syntax, agreeing to just change (keep, really) the terminology to “interface”. Drop “abstract” in the description of these types. There is general agreement with this sentiment.

Tucker wonders if there is there a problem with generics? He points out that Unchecked_Deallocation has roughly the following specification:

```
generic
      type Desig (<>) is limited private;
      type Ptr is access Desig;
```

It is clear that this has to support an access type declared as:

```
type Access_Deque is access Deque'Class;
```

or users will be very unhappy.

Erhard asks about generic derived types:

```
generic
      type Form is new Stack with private;
```

Tucker answers that Form only matches non-interface types because of the **with private**. The following could be used instead:

```
type T is abstract new Stack; -- For passing interface types.
```

The proposal also has a generic formal interface type:

```
type T is abstract;
```

which would be useful in:

```
generic
      type T is abstract;
      type Q is abstract new T;
```

Pascal says that he doesn't like the **is null** capability proposed in the AI. Tucker responds that this allows defining a "hook" in an interface, which you might not want to override. Pascal suggests that the **is null** and finalization stuff be moved to a separate proposal because it is independent of the rest of the AI, and it seems much less useful.

Various suggestions are made for Tucker. More should be said about generics. The terminology should be changed to "interfaces". An example like that given at the meeting should be added to the proposal. Other implementations should be mentioned.

Interface'Class should match (<>) indeterminate generic formals. Someone notes that Ada doesn't have an abstract untagged type, but that interface types are untagged. Tucker and Pascal note that a 'Class of an abstract type is not abstract. Some disbelief of this fact is muttered. Anyway, 'Class of an interface must be tagged for the semantics to work. So there is some opinion that such a type should not match formal abstract private types; they should only match tagged types.

There was an unanswered question on the interaction of the proposal with the **with type** proposal (in case it survives after all).

Tucker notes that more work is needed on the proposal to decide the properties of interface types. It seems that an interface type is abstract, but not tagged. On the other hand, interface'Class is not abstract, but is tagged!

Tucker will refine the proposal.

AI-252 Tagged Types, Object.Operation Notation, Object-Oriented Programming

The proposed notation is just a transformation for:

```
P.Print(Obj,...) → Obj.Print(...)
```

with added visibility semantics, i.e., there is no intent to create more elaborate rules on finding the controlling parameter in the signature of the called subprogram.

Tucker shows the following example:

```
package P is
      type T is tagged.…
      procedure Print (O : T'Class; …);
      function Image (O : T) return String;
end P;

with P;
package P2 is
      type T2 is new P.T with …
      function Image (O : T2) return String;
end P2;

package body P is
      procedure Print (O : T'Class, …) is
      begin
            Put_Line (Image(O));
      end Print;
      …
end P;
```

With these definitions, you would be able to write:

```
X : P.T;
X2 : P2.T2'Class;
…
```

```
X.Print (…);
X2.Print (…);
```

This notation is usable even for non-primitive subprograms.

Erhard asks: Won't you have a lot of ambiguity? What is the type of Obj in Obj.Print? This is not a problem, because only one operation can match. Tucker comments that the semantics are equivalent to “use”ing all of the packages in the derivation chain.

Now consider the following variant of the original example:

```
      with P;
      package P1 is
            type T1 is new P.T with …
            procedure Print (O : T'Class; …);
            …

--

      with P1;
      package P3 is
            type T3 is new P1.T1 with …
            function Image (O : T3) return String;
            …

--

      X3 : P3.T3;
      …
      X3.Print(…);
```

In this example, X3.Print () would find P1.Print ().

Pascal asks how the prefix is resolved. In particular, what if the prefix is a function? Tucker replies that this would be done the same way as currently.

Someone asks whether operator symbols are allowed. Tucker replies that of course they are. `Obj.”=“(`**`null`**`)` would test if some access is null. Tucker thinks that this even might reduce the use of **use type**; he opines that this is better than regular operator symbol calls, because at least the operator is between the operands. This draws general laughter from the ARG.

The object is always the first parameter. Pascal doesn’t like that. He would like to be able to use the first controlling parameter. That would be more complicated without much benefit. (Besides, this notation also works on untagged types, which have no controlling parameters.) The call syntax in the presence of defaulted parameters is particularly problematic, since you would have to “skip” some parameters.

Erhard finds it weird that we drag operations in that are declared in all the ancestor packages, as well as the class-wide operations which apply to the ancestors. What is the conceptual model of this feature? The model usually (in other languages) is that you are dot-selecting methods declared with the types. However, the proposed feature brings in other things (specifically, class-wide operations, but also other operations in the respective packages which are neither primitive nor class-wide). Tucker suggests thinking of the model as having implicit declarations of class-wide routines. Erhard agrees, saying that it is possible to think of class-wide operations as primitive, inherited but not overridable. Tucker notes that this is essentially “final” in Java or non-virtual in C++.

Erhard wants to consider the interaction with **use** clauses, and he puts the following example on the board:

```
with P;
package Random is
      procedure Print (O : T'Class);
      …
end Random;
```

```
use Random;

Obj.Print(...); -- Does this consider Random.Print? Tucker says no.
```

Should we also consider "used" packages in the resolution of Obj.Op? This could cause unnecessary ambiguities. Someone suggests using a preference rule to allow this to work, but it is immediately pointed out that the preference rules cause the infamous Beaujolais effect. Randy points out the users don't want to write the **use** clause anyway, so there is no point to such a preference rule. So it seems best to ignore the "used" units when resolving Obj.Op.

Erhard asks if this notation applies in the defining package itself? Tucker replies that it does.

Someone asks if the notation applies to visible items declared in the private part? The model is use-visibility, so it does apply.

There is possibly an issue due to the fact that overriding may change the defaults and the parameter names, so we need to decide which defaults and parameter names we use. Again, the use visibility model appears to provide the answer.

Steve B. asks whether this works if the tagged type is declared in the package body. In that case, we can't use use-visibility to describe it. So Tucker suggests a different description of the workings of the notation: This *adds* the ancestor chain to the visibility set. That is, it is like a block with a use clause for all of the ancestor packages, and then normal visibility (that is, it includes direct visibility). The ARG runs out of energy to discuss this issue; we decide to use this description as a working hypothesis.

We now turn to access type issues with this notation. Tucker demonstrates that the following call using the new notation:

```
      Acc_Obj : <Some named access type designating T'Class>
      Acc_Obj.Print (...)
```

would be equivalent to both:

```
      use Acc_Obj'Pkgs; -- All ancestor packages.
      Print (Acc_Obj,...)
```

and:

```
      use Acc_Obj.all'Pkgs;
      Print (Acc_Obj[.all],...) -- Possibly access parameters.
```

In other words, access types also consider implicit dereferencing, which can possibly introduce ambiguities.

Erhard starts a full example, to better understand how access types work in this context:

```
package P1 is
      type T1 is ...
      procedure Process (P : access T1);
      type AT1 is access T1'Class;
      procedure Process2 (P : AT1);
end P1;

with P1;
package P2 is
      type T2 is new P1.T1 with ...;
      procedure Process (P : access T2);
      type AT2 is access T2'Class;
end P2;

declare
      Obj : At2;
```

```
begin
      Obj.Process;  -- This works.
      Obj.Process2; -- This does not work. (Different access types.)
end;
```

John asks if we use this notation, can we omit the **with** clauses? Tucker answers yes, that is the value of the notation. Any routine that would be visible this way must be in the transitive closure of **with**s. In other words, this proposal reduces the number of **with** clauses that you have to write, but it doesn’t introduce implicit semantic dependencies.

Tucker notes that the prefix is a name, not an expression.

Erhard notes that if the prefix is **null**, the dereferencing raises Constraint_Error. Steve B. comments that this is just alternative syntax for an existing programming mistake (the existing syntax would contain an .**all**, which would raise Constraint_Error, too).

Randy asks that the AI's title be fixed, since the notation doesn't apply only to tagged types!

Tucker jokes that we don't need infix notation at all if we have this notation: If A is an Integer, it becomes possible to write:

```
      if A.”-”.”/=“(3).”not” then …
```

[Translation: `if not (-A /= 3) then`…] Someone notes that we now can have obfuscated Ada contests, just like C-family languages. This closes the discussion with laughter.

Tucker will revise the AI.

AI-257: Restrictions for implementation-defined entities

Randy queries whether amendments approved by the ARG are considered language-defined. There is a consensus that they are.

Should these restrictions apply to the library as a whole, or only to the current compilation unit? Tucker thinks that the standard configuration pragma approach is fine. More discussion follows and someone raises the point that Restrictions require all compilations to have the same restrictions values. This is true in order to handle runtime selection.

Randy comments that pragma Profile (AI-249) handles runtime selection better than random collections of Restrictions; shouldn't Restrictions no longer be required to be partition-wide ? Several people agree, but Joyce raises the bugaboo of compatibility.

Steve M. reads RM 13.12(8) out loud, noting that we can define these restriction arguments to not be partition-wide, because the paragraph says “unless otherwise specified”. There is general agreement to do that.

Tucker asks if No_Implementation_Pragmas includes implementation-defined pragma arguments? It is noted that some implementation-defined arguments are common (for instance, the convention “StdCall” on Windows). The consensus is yes, arguments are included.

Therefore, we should remove No_Implementation_Restrictions, because it is covered by No_Implementation_Pragmas (applied to pragmas Restrictions), whose definition forbids implementation-defined pragma arguments, too.

Approved intent 8-0-0. Randy will revise.

AI-260: How to replace S'Class'Input with custom versions

The proposal was submitted by Pascal Obry. After a bit of discussion, the proposal was modified to adding a pair of additional stream attributes for tagged types. The reason is that the proposal, 'External_Tag'Read, doesn’t fit within the existing syntax rules, and we don’t want to change the syntax if we have a good alternative. These attributes, 'Tag_Read and 'Tag_Write, can be used to set the tag I/O routines.

Steve B. suggests that this can only be usefully used on non-derived types. That's because the whole tree should be read/written the same way. Tucker says that such a restriction is not necessary. Randy points out that such a restriction is harmful, as it would prevent using this feature for controlled types. Tucker adds that this also applies to other types not in your control.

These are operational attributes, which are inherited.

Erhard points out that these can be used to control representation of tags for other reasons, such as shortening the representation of tags.

Randy will revise. Approved intent 7-0-1.

AI-261: Extending enumeration types

A short discussion occurs about the value and applicability of this proposal. Randy says that he wrote up the proposal mainly to capture all of the design work that was done in the e-mail discussion. He believes that all of the problems brought up in e-mail have been solved.

Tabled for now until people have a chance to read the proposal.

AI-262: Access to private units in the private part

Tucker shows that the syntax ought to be **private with**.

```
      with a, b, c, d;
private
      with e, f, g, h;
package P is
```

(Note that this is a suggestion, not the recommended style.)

There is some discussion as to whether this clause should go at the beginning of the private part or at the beginning of the unit. The argument made during the email discussion (that this would unnecessarily impact tools which peek at the beginning of the unit) appears compelling. So this clause should go at the beginning of the unit.

private needs to be given with each keyword **with** to which it applies. For instance:

```
      with a, b, c, d;
      private with e, f, g;
      with h;
      private with i, j;
      package P is …
```

Randy will complete the write-up.

Approve intent: 7-0-1.

AI-264: Exceptions as Types

Erhard summarizes the results of the Exception Workshop at Ada Europe. There is a desire for exceptions as first-class citizens in the language. The workshop participants definitely would like a hierarchy. They also would like more structured user-definable information associated with exceptions. Further, they would like to be notified when an exception terminates a task (possibly a routine that would be called, with the exception id and task id as parameters). In order to achieve this notification, the notion of task groups à la Java thread groups was mentioned. The main point in this is that some other task, most likely the creator task, should be able to specify the action to be taken if the created task terminates with an exception. Some wanted exceptions as part of a (subprogram) signature, again with Java as the example.

Pascal wonders if his proposal is too complex, as pointed out by Randy in an email discussion which can be found in the appendix of the AI. Adding a new class of types to the language is certainly a major change. Moreover, the "exception types" could contain components of any type (controlled types, protected objects, even tasks!), which seems unreasonable. Also, it is unclear how this could be made to work compatibly with the existing language. Especially annoying is the fact that there is currently only one type, Exception_Occurrence, to represent all exceptions, but the proposal would have one occurrence type per exception.

Tucker suggests a simpler proposal than the one in the AI: Raise_Object, where the object is a tagged object of a special type, which is non-limited. This type cannot have any controlled components. This suggestion doesn't draw much of a reaction from the ARG.

Erhard comments on the AI proposal. He does not believe that the exceptions handlers should be required to be disjoint. For instance, consider that you might want to handle Name_Error and then (separately) all of the rest of the I/O exceptions. You certainly don't want to be forced into listing all of them.

A suggestion is made. Don't require disjointness, but rather require the handler for the more-specific exception occur before a less-specific exception. This is similar to the way that **others** is handled. There is general agreement that this is OK (cut and paste errors do not cause problems, as they do in a purely positional scheme).

Erhard comments that he would like exceptions as part of signatures. This would include static checks that the list of exception is correct. There is not much support for this idea. Erhard takes an action item to write up a proposal. Tucker wonders about the value of this proposal, he would like to know what people really use exceptions for. Others agree. Erhard withdraws his action item, seeing no support.

Pascal will revise the AI with a simpler proposal.

AI-265: Task Activation Policy for High-Integrity Systems

The question was raised as to why failure is a bounded error. The answer is that it might be a distributed overhead.

Tucker suggests referencing 10.2(30) for failed activation; this paragraph states that you can kill everything if activation fails. In the context of this AI, activation takes places at the **begin** of the environment task, and there can be no handler there, so the environment task aborts, so any dependent task aborts. Therefore, the last implementation advice of the AI is vacuous.

John and Ben suggest that the policy identifiers have questionable connotations: Standard looks "good", which makes Deferred look "bad". Someone suggests using Default instead of Standard; another suggestion is to use Regular.

Tucker suggests changing both names: "library-level" vs. "environment-level". A wealth of suggestions is triggered: Per_Library_Unit vs. Per_Partition, Per_Unit vs. Per_Partition, Per_Package vs. Per_Partition.

The discussion briefly turns to a technical issue: the write-up needs to be careful to allow other (non-library) tasks to exist, because this pragma could be used separately from the Ravenscar profile. The conclusion is that this pragma only affects declared library-level tasks.

Steve M. returns the discussion to naming, suggesting that the pragma be named Library_Level_Task_Activation_Policy. This receives little support because it is too much of a mouthful.

Brian suggests changing the pragma name to Partition_Elaboration_Policy, with policy identifiers Sequential and Concurrent. The default would be Concurrent, which represents the current semantics. General agreement.

Joyce and Brian will update AI-265.

AI-266: Task Termination procedure

Erhard comments that this capability could be coded today explicitly, in part by using an **others** handler. However, a language use regulation (no exception handlers) prevent that approach. To add extra language seems a bit heavy for a situation that could be expressed already. Randy notes that this feature also can be written in Ada 95 using a limited controlled type.

Should the termination procedure only be called on exceptional termination? Brian answers No, it is intended to cover any termination. Ravenscar has no termination.

At the request of several members, Brian gives a history of this feature. Essentially, it was created so that Ravenscar programs (which do not allow task termination) can be notified if a task terminates for any reason. They would have preferred to ban termination, but of course that is impossible.

Tucker thinks that the AI's authors have gone astray. Their needs don't seem to really be per-task handling of termination. [Editor's note: the problem statement in the AI doesn't seem to reflect the above explanation, either.]

Tucker would want to catch the silent death of a task in the *master* of the task which is dying, and he would also like to put the pragma-or-whatever in the master.

Erhard points out the difference to the Workshop proposal, which wants the temination procedure on exceptional termination only and specifyable "from the outside" of the respective task. A possible suggestion are operational attributes Task_Obj'Finalize and Task_Obj'Exceptional_Finalize or suchlike.

The conclusion is to send this proposal back to the authors; a liaison with the Exception Workshop is suggested.

Joyce and Brian will try to come up with a new AI-266.

On Saturday, Tucker makes a mini-proposal to handle the AI-266 problem.

He proposes that type Root_Task_Group be declared somewhere with a set of useful operations, then a user can declare a task group object.

```
TGObj : Root_Task_Group'Class;
pragma Task_Group (TGObj
                  [, On => Task or TaskType or Acc_to_Task_Type]);
-- If no ON part, applies to all the dependent tasks of this master.
```

The following operations would then be called by any task in TGObj when the named condition arises:

```
Aborted (TGObj, TaskID);
Unhandled_Exception (TGObj, TaskID, Exc_Occ);
Normal_Termination (TGObj, TaskId);
```

The pragmas has to be given before the task activates, so it can only be at the place of a declarative item. Possibly one could allow pragma Task_Group to appear inside a task type spec, then discriminants can be passed in, much like what we have for Priority and Storage_Size.

Erhard suggests alternative semantics, with an "implicit" defaulted discriminant on all tasks that one can specify if needed (this being the Java model). Hooted down by the group.

Tucker is given an action item to write this up.

AI-267: Fast float to integer conversions

Tucker thinks the attribute should return a float. Name the attribute Machine_Rounding:

```
      function S’Machine_Rounding (X : T) return T;
```

Compilers will have to recognize the pattern Integer (F'Machine_Rounding(X)) to generate better code.

This attribute is static if its prefix and its argument are static, so should there be a requirement that static and dynamic evaluations give the same result?

Pascal argues no. Randy says the proposal specifically leaves it unspecified, no one is supposed to care.

Randy will revise.

Approve intent: 6-0-0.

Detailed Review of Regular AIs

AI-185: Branch cuts of inverse trigonometric and hyperbolic functions

No one at the meeting other than Pascal understands this enough to comment on it. Mike and Pascal have reworked the AI, and we trust them to have gotten this right. John says he’s convinced by the reference to Kahan, an authority in numerics.

Approved: 5-0-3

AI-195: Streams

Tucker has not worked on the stream size proposal. It is suggested that the stream size proposal (item 10) should be handled as an amendment - it originally was “sneaked in” when the ARG was not considering amendments.

Randy is directed to split the AI. Tucker will be responsible for the stream size amendment. Pascal will revise the remainder of the AI for consideration at the next meeting.

AI-229: Accessibility rules and generics

Tucker suggests merging the existing wording sentence with the new one. Pascal notes that this would be a bigger change, as it would disallow 'Access in the specification of a generic.

John then wonders why we don't have this problem with objects. Tucker notes that there is a runtime check for objects. He then wonders why runtime checks do not apply to access-to-subprogram; he thinks it has something to do with generic sharing.

Randy is asked about the generic sharing problem that led to the original rule. He explains that the problem is that the “hidden” instantiation data parameter used for all routines declared in a generic unit cannot be added when 'Access is used in the body (even through a “thunk” could be generated at that point, there is no way to identify the appropriate data when calling through the access-to-subprogram object.) This is not a problem for access types declared in the generic unit as the parameter is included for such types. Nor is it a problem for 'Access used in the generic specification, as the needed “thunk” can be created when the generic is instantiated.

Tucker considers relaxing the existing rule to merge it with the new one, but there seems to be little interest.

Eventually, the rule as proposed is retained.

Pascal notes that the new sentence should say “ultimate ancestor of S”.

Approval with changes: 6-0-2

AI-237: Finalization of task attributes

The changes discussed at the last meeting have been incorporated.

The summary ought to mention that Current_Task is unusable during finalization of the attributes.

Tucker points out that the permission to finalize is **only** to finalize earlier than the instantiation is finalized, not later. Joyce suggests a rewording, which Tucker refines. Move this insertion after C.7.2(28).

> If the task terminates before the master of the instantiation of Task_Attributes is finalized, an implementation may finalize the object corresponding to the attribute of the task at this time (instead of when the master of the instantiation is finalized), and reclaim any other storage associated with the attribute.

The race condition is addressed by “instead of” in the above paragraph.

Approval with changes: 8-0-0

AI-253: The legality rules for pragmas Attach_Handler and Interrupt_Handler are similar

No discussion.

Approval: 8-0-0

AI-255: Object renaming of subcomponents of generic in out objects

Steve B. worries that making the subtype unconstrained grants permissions as well as eliminates them. Steve and Tuck discuss this over a break and determine that there is no problem, because formal objects are not aliased.

Delete the “Why? (Tuck said so.)” from the question.

The group would like to improve the discussion to explain why the change is necessary, rather than leaning on compatibility with Ada 83. An example showing a disappearing component would be compelling.

Pascal will revise.

Approve intent: 8-0-0.

AI-258: Behavior of Interfaces.C.To_C when the result is null

Erhard remarks that of the three possible “solutions”, only the selected one makes sense. Randy made a compiler review and it appears that just about everyone returns 0 .. Integer’Last or some similar absurdity.

Remove the “[Editor's note:”.

Approve with editorial changes: 7-0-1.

AI-263: Scalar formal derived types are never static

Split the other problem out into a separate AI.

Erhard doesn't like making these things non-static because staticness is useful. But it is clear that the language is broken and that implementations depend on the invariant that static values in the generic are merely copied to the instance.

Change wording: "is not a descendant of a scalar formal type" $\rightarrow$ "is not a descendant of a formal type".

Much consideration is given to whether the problem also occurs for static string types. Pascal claims that static string types can't have the problem because only a few attributes are static (First, Last, Length), and they are part of the contract. Indeed, the rule is too restrictive for those, it isn't necessary to define them to be non-static. Therefore, Tucker and Pascal would like to drop the formal array restriction.

Approve the intent to make derived and non-derived types the same. 6-0-2 (Joyce, Steve M. abstain)

Steve Baird will revise, determining whether formal arrays should be included.

AI-268: Rounding of real static expressions

Erhard brings up a meta-issue: What is the feeling about mandating IEEE support? Or as an implementation advice? There is not much support for *mandating* IEEE support (as this would prevent Ada from being used on some targets on which it is current used.) Pascal comments that IEEE attributes would be useful to support NaNs, infinities, rounding modes, and the like. Several people note that the ARG doesn't have much expertise in numerics issues; we need to get others involved with these decisions, preferably some IEEE person (apparently Erhard knows such a person).

Returning to the AI itself: Change ACATS testing to say that is it is implementation-advice and thus untestable.

Approved with change: 8-0-0.