Minutes of the 19th ARG Meeting

20-22 June 2003

Toulouse, France

Attendees: Steve Baird, John Barnes, Randy Brukardt, Alan Burns, Gary Dismukes (all but Sunday afternoon), Kiyoshi Ishihata, Pascal Leroy, Steve Michell (Friday and Saturday only), Erhard Ploedereder, Jean-Pierre Rosen, Tucker Taft, Joyce Tokar (Friday only), Tullio Vardanega (Friday and Saturday through afternoon break only). Observers: Ed Schonberg (Friday and Saturday only), Arnaud Lecanu (Friday only).

Meeting Summary

The meeting convened on 20 June 2003 at 14:00 hours and adjourned at 15:15 hours on 22 June 2003. The meeting was held in a conference room at the Mercure Atria Hotel, in Toulouse, France. The meeting was hosted by the Ada Europe conference on Friday, and by IBM Rational Software (Pascal Leroy) on Saturday and Sunday. The meeting covered most of the amendment AIs on the agenda, but no normal AIs from the agenda.

AI Summary

The following AI was removed from the hold state (it was previously approved):

    AI-298/04 Non-Preemptive Dispatching

The following AIs were approved with editorial changes:

    AI-297/05 Timing events (10-0-3)
    AI-321/03 Definition of dispatching policies (13-0-0)
    AI-326/01 Tagged incomplete types (10-0-1)
    AI-328/01 Add preinstantiations of Complex_IO (9-0-0)

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

    AI-217-6/02 Limited with clauses (10-0-2)
    AI-218-3/01 Accidental overloading when overriding (9-1-1)
    AI-230/05 Generalized use of anonymous access types (10-0-0)
    AI-231/03 Access to constant and null-excluding access subtypes (8-0-1)
    AI-251/07 Abstract interfaces to provide multiple inheritance (8-0-1)
    AI-252/03 Object.Operation notation (5-1-3)
    AI-285/04 Support for 16-bit and 32-bit characters (9-0-1)
    AI-296/02 Vector and matrix operations (7-0-3)

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

    AI-237/05 Finalization of task attributes
    AI-266-2/03 Task termination procedure
    AI-292/00 Sockets operations
    AI-327/01 Dynamic ceiling priorities

The following alternative AIs were discussed and put on hold as another alternative is pursued:

    AI-217-5/02 Type stubs with limited context clauses
    AI-217-7/02 Incomplete type completed in a child
    AI-218-2/04 Accidental overloading when overriding

Detailed Minutes

Meeting Minutes

We consider the minutes of the 18th ARG meeting. The minutes were approved by acclamation.

Publicity of the Amendment

We need more articles for the Ada User Journal; they are prepared to publish one for each issue (every quarter). John Barnes volunteers to do an article on the vector and matrix packages for the next issue (October).

Pascal is working on an update of his overview for Ada Letters.

Next Meeting

As decided at meeting 18, the next meeting will be hosted by Steve Michell in Sydney, Nova Scotia, October 3-5, 2003.

We then discuss the following meeting. SIGAda is too close to the October meeting (December), so we need a host for a February meeting. Joyce Tokar volunteers to host a meeting in the Phoenix area. She will need to find facilities, and suggests that, as February is the busy season, we select dates now. Tucker says he is not available February 15-21. We select dates of February 27-29, 2004, and Joyce is given instructions to suggest alternative dates if necessary to find appropriate facilities.

The meeting after that will probably be held after Ada Europe (June 18-20, 2004, Palma de Mallorca, Spain)

Motions

The ARG thanks our hosts, Ada Europe and IBM Rational Software, for hosting the meeting. Approved by acclamation.

The ARG thanks our emergency minute taker, Tucker Taft for volunteering for the task of recording minutes when Randy Brukardt's laptop failed 2 1/2 hours into the meeting. [And especially for putting up with Randy reading over his shoulder and suggesting items to record for the rest of the meeting. - Editor]. 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-294

John Barnes:

- AI-204
- AI-218-2
- AI-296

Randy Brukardt:

- AI-217-5
- AI-279
- AI-280
- AI-320
- Create an extended example of the use of AI-217-05, etc.
- Create a detailed design of the implementation of AI-217-05, AI-217-06, AI-217-07 in Janus/Ada.

Editorial changes only:

- AI-167
- AI-178
- AI-216
- AI-224
- AI-228
- AI-256
- AI-259
- AI-265
- AI-270
- AI-283
- AI-287
- AI-306
- AI-310
- AI-316

Alan Burns:

- AI-266-2
- AI-297
- AI-307
- AI-321
- Write an article for Ada User Journal on real-time features of the Amendment.

Gary Dismukes:

- AI-158

Pascal Leroy:

- AI-217-6
- AI-285
- Create a detailed design of the implementation of AI-217-6 in Apex.

Tucker Taft:

- AI-217-7
- AI-230
- AI-231
- AI-252
- AI-318 (was assigned to Bob Duff)
- Create a detailed design of the implementation of AI-217-6 and AI-217-7 in AdaMagic.
- Present AI-217 examples to Ada UK, report on comments (was assigned to John Barnes).

New Action Items

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

Steve Baird:

- AI-51
- AI-109

- AI-251
- AI-291

John Barnes:

- AI-254
- AI-296

Randy Brukardt:

- AI-218-3
- AI-301
- Update Janus/Ada implementation report for current version of AI-217-6.

  Editorial changes only:

- AI-297
- AI-298
- AI-321
- AI-326
- AI-328

Alan Burns:

- AI-266-2
- AI-327

Bob Duff:

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

Kiyoshi Ishihata:

- Consult with the Japanese SC22 about the acceptability of AI-285.

Mike Kamrad:

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

Pascal Leroy:

- AI-217-6
- AI-264
- AI-285
- AI-311
- Write an article on the Amendment for the Ada Letters.

- Write a report on access subtypes (with Tucker Taft).
- Update Apex implementation report for current version of AI-217-6.

Steve Michell:

- AI-148
- AI-250

Erhard Ploedereder:

- AI-237

Ed Schonberg:

- AI-292
- Create a GNAT implementation report for current version of AI-217-6.

Tucker Taft:

- AI-133
- AI-162
- AI-188
- AI-214
- AI-230
- AI-231
- AI-252
- AI-275
- AI-282
- AI-288 (split into two AIs: pre/postconditions and invariants).
- AI-290
- AI-293
- AI-295
- AI-304
- AI-312
- AI-317
- Update AdaMagic implementation report for current version of AI-217-6.
- Write a report on access subtypes (with Pascal Leroy).

Mike Yoder:

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

Items on hold:

- AI-284 (waiting for more keywords to be defined)

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-217-5/02: Type stubs with limited context clauses
AI-217-6/02: Limited with clauses
AI-217-7/02: Incomplete type completed in a child

Tucker gave a report from the AdaUK presentation:

- Type stub was considered less desirable than the other options (it is more work to use than the others).

- Incomplete type completed in child more familiar; it was the favorite of the audience.

- **Limited with** appeared to be more work, in particular, it creates a new kind of compilation dependence (of course, that is true of all of these proposals, it just isn't so obvious to users).

Randy asks if the limitation that instantiations cannot be accessed via a **limited with** is going to be a problem. In the past (for instance, child units), we had to add generic versions of features due to user demand.

Pascal is not convinced this is a real limitation; he tried but was unable to come up with any example where it would matter. Remember that these are intended to be used to support mutually recursive types; how can two instantiations have mutual recursion?

Steve Baird suggests a user that could derive from the type defined by an instantiation and use that to be the incomplete type accessed by a **limited with**.

What do you get when you do a **limited with** of an instantiation or a renaming? You get an error if you directly name an instantiation or renaming, you do not not see it if it is nested (which probably generates an error at the point of the intended use).

What about renamings that are used to select one of two implementations? Those seem unlikely to require a **limited with**.

Incomplete tagged types will be supported by all proposals; it is now a separate AI (AI-326), which we'll discuss later.

Pascal comments that type stubs and incomplete types completed in a child defer some error checking to link time; **limited with** performs checks at compile time. Generally, it is better to detect errors earlier.

Library-based compilers will have more trouble with **limited with** due to new kinds of dependence.

We take a series of straw votes on the three proposals.

- Prefer **limited-with** vs. something else: 6-1-5.

- Prefer type-stubs vs. something else: 0-7-6.

- Prefer incomplete type completed in child vs. something else: 0-6-7.

Based on this vote, we decide to proceed with **limited with** only; the other two proposals will be put into the deep freeze of discarded alternatives.

We turn to discussing the details of the **limited with** proposal.

Pascal asks why the first sentence of the 10.1.1(26) insertion is needed (this originally was worded by Bob Duff). We want

```
limited with A.B.C;
```
to mean
```
limited with A, A.B, A.B.C;
```
and that's what this rule does. So we need this rule.

What is the effect of **limited with** in scope of a normal **with**? Consider:

```
with Q;
package P is
    use Q;
    ...
end P;

limited with Q;
package P.Child is
```

We probably should disallow **limited with** on bodies; it's useless and causes ugly corner cases. Jean-Pierre comments that this sends a clear message that this is for breaking circularity and not for other uses.

If a child has a **limited with**, it overrides the normal **with** inherited from its parent package.

What happens when using a rename of the package found in a second package that had a "normal" **with** of the package? Consider:

```
package Q is
    type T is ...
    procedure Proc (Param : T);
    X : T;
end Q;

with Q;
package P is
    package R renames Q;
end P;

with P;
limited with Q;
package PP is
    Obj1 : Q.T;   -- Illegal (Q.T is incomplete)
    Obj2 : P.R.T; -- Legal??? (No.)
end PP;
```

The rename sees a limited view of the package, so Obj2 is illegal.

What happens when using a subtype or object found in a second package that had a "normal" **with** of the package? Consider:

```
with Q;
package R1 is
    subtype Foo is Q.T;
    type Rec is private;
    Z : Q.T;
    function "=" (A, B : Q.T) return Boolean renames "=";
private
    type Rec is
        record
            F : Q.T;
```

```
                    end record;
        end R1;

        with R1;
        limited with Q;
        package S1 is
            X : R1.Rec; -- Legal? (Yes.)
            Y : Q.T;    -- Illegal (Q.T is incomplete)
            Z : R1.Foo; -- Legal? (Yes.)
            Maybe : Boolean := R."=" (R1.Z, X.F);
                 -- Legal? (Yes, if F is visible.)
        end S1;
```

The package sees the second package's (R1's) view of the subtype or object. Ed notes that Q.T and R1.Foo are the same type; it appears that we have two views of the same type. This would very hard to implement in ACT's compiler. Much discussion ensues.

Pascal notes that his idea was that the type would be incomplete in all views. But that means breaking privateness; why would a user expect R1.Rec to be incomplete? It doesn't visibly depend on Q.

It is suggested that the view is a property of the expanded name, and not of the type. An expanded name giving the limited view (such as Q.T) could carry a "flag" even though both names (Q.T and P2.S) point at the same entity. Ed thinks that might work.

Could we live with the ripple effect? It seems to be impossible to "hide" it in diamond cases like that above. We'd rather not; ripple effects are very bad for users as they cause no end of trouble during maintenance. Besides, if we had no rules on the use of **limited with**, we could have a real circular dependence and still be legal if all of the units involved used **limited with**. That obviously wouldn't work. So we need some restrictions on **limited with**, and we might as well eliminate ripple problems at the same time.

John expresses concern that the added complexity of **limited-with** makes him more interested in the incomplete-type-completed-in-a-child. This alternative does not share all the same complexity because there is a separate source construct for the incomplete type declaration, and the full type always sees it. However, others prefer the type stub approach if we give up on the **limited with**, so we would be back in the stalemate.

Erhard wonders if we are trying too hard, and should just allow circularity. That way lies madness; the linear elaboration model would have to be abandoned. Erhard withdraws his suggestion.

These problems get worse if we can have procedure calls. So, a **limited with** clause should not be allowed on a body or a subunit.

How does this interact with **private with** (AI-262)? Textually, **limited private with** makes sense. **private limited with** would also make sense, but preserving keyword order favors the first one.

Erhard wonders if we need **private** at all in context clauses. He is reminded that AI-262 was extensively discussed and is already approved (and thus it is out of order to suggest changing it), but he insists on making the point that **private with** does not need to be syntactically distinguished, we could simply say that private children are only visible in the private part. A mercifully short discussion ensues. Most felt that from a documentation point of view, it is useful to distinguish the dependencies that are only for the private part. **Private with** complements the **private** child concept. This is covered in the AI's discussion section, and was discussed extensively by e-mail recorded in the appendix.

We decide that **limited private with** would be allowed, with semantics similar to that defined in AI-262.

When is the type incomplete? When can you dereference an access-to-incomplete? This is complicated enough that we need an example:

```
        package Q3 is
            type T is tagged
                record
```

```
            Comp : Natural;
            ...
        end record;
    procedure Proc (Param : T);
    X : T;
end Q3;

limited with Q3;
package P3 is
    type Acc_Inc is access Q3.T; -- Always an access-to-incomplete,
                                 -- for ever and ever.
    X : Acc_Inc;
end P3;

[limited] with Q3;
with P3;
package R3 is
    M : Q3.T := P3.X.all; -- OK if not limited with.
    procedure Proc (X : Q3.T);
end R3;

with P3, R3;
procedure S3 is
    N : Natural := P3.X.all.Comp; -- Illegal.
begin
    R3.Proc (P3.X.all); -- Illegal.
end S3;
```

The suggested rules are "A dereference of an access-to-incomplete is permitted anywhere, but is incomplete unless in the scope of a normal **with** clause for the library package (or a library renaming thereof) in which the full type is declared. An incomplete type always matches the corresponding complete type."

Without a rule like this, we have a ripple effect; changing R3 from with to limited with would cause S3's dereferences to become legal. Ripple effects are nasty for maintenance and for incremental compilations.

Jean-Pierre wonders if a full **with** should be required on the body of a unit with a **limited with**. There doesn't seem to be any benefit to requiring that; what if there isn't a body? The suggestion is dropped.

Randy wonders if you can **limited with** a unit that you are not allowed to **with** (because it is **private**)? No, the same rules apply here that apply to normal **with**. It could be a **limited private with**. Whether a unit is a private child can be determined syntactically, so there is no problem. Randy notes that this is different than most of the other proposals, where it is hard or impossible to disallow this (we don't have the information).

Jean-Pierre wonders if we need an Implementation Permission to refuse a **limited with** if the unit hasn't been added to the environment. We certainly don't want to require that it always be legal; the source must be available (for source-based compilers) or the unit pre-compiled (for library-based compilers). Tucker suggests that it simply be implementation-defined. Erhard suggests "It is implementation-defined how a compilation unit mentioned in a **limited with** clause is entered into the environment.", which meets with general approval.

Steve Baird asks if a unit that is only mentioned in a **limited with** clause is needed in the partition. Tucker suggests yes. Erhard does not agree initially, but ultimately is convinced. Pascal thinks that a post-compilation rule would be needed to reject the partition if there is no full view. Tucker says that is not necessary, it is needed, it gets elaborated somewhere (which is not a problem, as a limited view has no effect when elaborated). But we would then have to define the meaning of elaborating a limited view. It would be better to simply add to 10.2: "If the limited view of a unit is needed, then the full view of the unit is needed." Jean-Pierre thinks this is good: if during development, we only need the limited view for a while, the effect does not suddenly change when the full view becomes referenced.

Tucker recommends replacing the new term "explicitly mentioned" with "named", in order to reduce confusion with the existing technical term "mentioned".

A use clause is not permitted on limited view of a package. Use clauses are not normally inherited, but they can be inherited from a parent unit. What if parent unit has a "use" clause of full view of package? The "use" clause for a package loses its effect within the scope of a limited view of the package.

```
with A.B.C;
package P4 is
     use A.B;
end P4;

limited with A.B;
package P4.C is
      -- A.B use visible here? (No.)
end P4.C;
```

For methodological reasons, we also disallow *naming* in a **limited with** clause a unit (or renaming thereof) that is *mentioned* in an enclosing normal **with** clause. This is independent of whether the **with** clause is **private**.

```
limited with A.B; -- Illegal.
limited with A.B.C; -- Illegal.
limited with A.B.C.D; -- Legal.
package P4.C2 is ...
```

Gary adds renaming to the example:

```
with A.B.C;
package R5 renames A.B.C;

with R5;
package P5 is
     use R5;
end P5;

limited with A.B; -- Illegal.
package P5.C is ...
```

In other words, use clauses are illegal for limited views and are not inherited in the places that see a limited view.

Approve intent of AI: 10-0-2

AI-218-2/04: Accidental overloading when overriding
AI-218-3/01: Accidental overloading when overriding

Randy explains that alternative 3 grew out of a discussion between him, John, and Pascal about a draft of alternative 2. Alternative 3 differs from alternative 2 by providing "overriding" and "not overriding" as overriding_indicators, eliminating the "maybe overriding" concept. It also eliminates the configuration pragma (it could be defined separately if we wish). This appears to simplify the proposal.

If we defined a configuration pragma, it would require an overriding_indicator on all primitives of derived types and primitive operators. This could be restricted to tagged types as an option.

This approach gets general agreement, and we decide to proceed with alternative 3 only.

We turn to considering details of alternative 3.

Should an overriding_indicator be allowed on non-primitives? No. Should an overriding_indicator be allowed on non-tagged type primitives? Yes (otherwise we could have contract problems with private types and generics).

What happens if a subprogram is primitive on two types? There is no problem, the overriding_indicator better be "not overriding", since a subprogram cannot possibly be overriding if primitive on more than one type(Editor's node: This statement was discovered to be false after the meeting, so some resolution will be needed.)

What about non-operator primitives of a root type? These can have an overriding_indicator, but of course the indicator had better be "not overriding".

How should these be checked in generics? In the generic unit? In the instance? In Both? Neither?

Two options were seriously considered:

1. Assume the best in generic declaration. This means:

   - **overriding** can't fail in generic declaration (only can fail in instance),

   - **not overriding** can fail in generic declaration, if we have a formal derived type and it is known to have a particular primitive (and instance),

   - Recheck in instance (including private part),

   - Check body based on properties of formal in body

2. Check in generic declaration based on properties of formal. This means:

   - **overriding** can fail in generic declaration (but it can't fail in instance),

   - **not overriding** can fail in generic declaration (and in instance),

   - Recheck in visible part and private part of generic instance

We take a straw vote comparing the two options: option (1) get 2 votes, option (2) gets 6 votes, 3 abstain.

If you give a **not overriding** specification, must it apply to any primitives that would be implicitly declared later within the immediate scope? Yes. We don't want contradictory clauses. If you give an overriding_indicator on the partial view the implementation has to recheck it when new operations come into existence. This implies that you cannot give an **overriding** after **not overriding**.

Where may an **overriding** clause be? Immediately within the immediate scope of the declaration, but how late can it be? Randy suggests that for checking purposes, it is better to restrict it to the specification. However, overriding can occur in the body. Perhaps we should allow overriding_indicators on the body. A separate overriding_clause must be no later than the body, if there is one. Perhaps we can eliminate the separate overriding_clause completely if we allow indicators on the body. But that might cause problems with instantiations.

Vote on intent: 9-1-1. Jean-Pierre votes against the proposal, saying that it is overkill.

AI-230/05: Generalized use of anonymous access types

Tucker explains the changes. Object declarations were dropped; object renaming was added as a replacement. Generic formal **in** objects also were dropped for consistency. Pascal complains that this causes a loss of orthogonality: **access** doesn't work for generic formal parameters. We take a straw poll to see if generic **in** objects should be retained: 0-4-6. (Interestingly, Pascal didn't vote for this either, so why'd he bring it up?)

Typos: in 3.8(18/1): compo{n}ent; in 3.10.1(12) is the same as {that of} the containing object

Wording for the special Is_Null attribute of an object of any access type needs to be added; see the February 2002 minutes.

Tucker raises a straw man of a membership operation "obj in null". Pascal comments that he likes that better, as it is more readable that obj'Is_Null. This causes a group groan.

Jean-Pierre asks about the equality of a record containing a component of an anonymous access type. There is no equality. Ouch!

Tucker argues that it would make more sense to make "/=" and "=" work than to invent yet more new syntax. After all, types in Standard already have universally visible operators. And that would provide a fix to Jean-Pierre's problem.

Could an operation be declared at the point of the anonymous type, hoisted to the enclosing scope? Yes, but that would sometimes end up ambiguous (presuming that convertible anonymous types could be compared, multiple anonymous types in the same scope would cause trouble).

So we are back to a universal equality in Standard, but at least one operand must be of an anonymous access type (name resolution), and the types of the operands must be convertible to one another (legality rule). **null** = **null** is now always illegal, since there is no context for resolving the type of **null**, which requires a single expected access type. The same goes for allocator = allocator, allocator = **null**, etc. This is a tiny incompatibility (**null** = **null**, etc. are legal if precisely one access type is visible), but the constructs are useless, so it isn't worth trying to fix them. This is very much like the fixed multiply/divide operators.

Tucker says (off-topic) that he thinks he has a solution for the fixed multiply/divide operator problems. He'll create an AI.

What is the storage pool of an anonymous access type? Tucker believes that the standard already covers that. The group feels that wording should be added to describe the storage pool, or at least a note to explain how wording in the existing standard already covers storage pool issues.

Correct the example, the end of the package body got cut off.

Tucker suggests that we need to address issue of functions returning anonymous access types; that is covered by a different AI (AI-325).

Approve intent of AI: 10-0-0

AI-231/03: Access to constant and null-excluding access subtypes

Should **access** T be equivalent to **access all** T **not null** (which is compatible), or **access all** T (which is consistent)?

Having an implicit **not null** is weird. The incompatibility is only that Constraint_Error is not raised at the call (it will be raised inside the subprogram); this thus is a performance incompatibility. Pascal expresses concern about this incompatibility. Note that for tagged controlling access parameters, there is an implicit dereference to get the tag, so there is no change (these are effectively null excluding subtypes, whether or not they are declared that way). For other types, the performance fix is simply to add **not null**. Steve Baird comments that we still need to add a rule indicating there is an implicit dereference for a tagged controlling access parameter so the designated type cannot be incomplete and so the language defines the check.

Tucker dropped **all**, since **all** is the same as nothing. Pascal and Randy object; they see this as similar to a named access type, and it would be odd to have to say **all** in a named access type, and not be able to in an anonymous access type. We can handle this similarly to **in** for parameters; if neither **all** nor **constant** is given, it is considered implicitly equivalent to **all**.

For a null-excluding subtype, the default initialization will produce a Constraint_Error. Someone asks if this situation could be avoided by disallowing default initialization of **not null** subtypes (you would have to always give an initial value). But this would lead to privacy and contract model problems. So we decide that raising Constraint_Error in this case doesn't seem to be a problem.

Is **not null** allowed on a parameter subtype? No, except as part of an access_definition.

In an access type definition, the subtype indication when the designated type is an access type is ambiguous.

```
type T is access D not null;
```

If D is an access type, to what type does the **not null** apply? As defined in the AI, it applies to the newly defined type, rather than D which is what would be expected.

Tucker suggests changing the grammar to:

```
access_to_object_definition ::=
    access [general_access_modifier] subtype_mark
           [scalar_constraint|composite_constraint]
  | access [general_access_modifier] subtype_mark not_null_constraint
```

Steve Baird notes that for constrained access subtypes there is an invariant that if a pointer is constrained the constraint check is performed on assignment, there is no way it can change behind your back. Discriminants of aliased objects cannot be changed, even by a whole-object assignment, since all aliased objects are constrained by their initial value. Tucker replies that this is not a problem, because the **not null** constraint applies to the value of the pointer, not the designated object.

Erhard is concerned that it will be confusing to which type a **not null** constraint applies in an access type definition. It could be either the designated type (if it is an access type) or the newly defined access type. Even through the language clearly defines it, it won't be clear from reading.

Pascal suggests moving the **not null** in front. Specifically:

```
type T is [not_null_constraint] access [all|constant] subtype_indication
subtype_indication ::=
    [not_null_constraint] subtype_mark
        [scalar_constraint|composite_constraint]
```

This would look like:

```
procedure P (Obj : not null access D);
procedure Q (Obj : not null access constant D);
subtype S is not null D;
type T is not null access all D;
```

Straw vote: Move **not null** in front of **access**/subtype-indication: 7-1-1. (Kiyoshi prefers **access not-null** D.)

Straw vote: Allow constraining the designated type in an access definition? (This would look like **not null access not null**): 7-1-1.

This would give us a new record for a sequence of reserved words:

```
... is not null access constant not null ...
```

Approve intent of AI: 8-0-1.

AI-251/07: Abstract interfaces to provide multiple inheritance

Steve explains changes.

Jean-Pierre asks why we need a separate syntax for these. That is, why do we need to call these **interface**s instead of just **abstract tagged null record**? There are a number of reasons:

- Conceptually different;
- Maintainability - we don't want interface-ness to be dependent on the absence of non-abstract/non-null primitives. (if that was true, a programmer could add a subprogram and silently change all of the semantics);
- Implemented in a different way;
- Different legality rules;
- Should be a first-class feature. We don't want to use a **pragma** to mark these; we don't have a **pragma** Class.

Tucker asks why we disallow private primitives if they are **null** procedures? Steve Baird gives a long answer, which is not clearly understood. Tucker agrees to take this discussion off-line.

Jean-Pierre suggests that we should allow **is null** in a generic formal subprogram default. This gets general agreement. We should also change Finalize/Adjust/Initialize to be **is null**. They're essentially defined that way anyway. Conclusion: Separate out **is null** into separate AI but assume *explicitly* in AI-251 that the **is null** AI will be approved.

Jean-Pierre asks whether **out** parameters should be permitted in a **null** procedure. Yes, though compilers might give a warning (at least for scalar **out** parameters and possibly by-copy composites).

Jean-Pierre continues by asking whether **is null** should be allowed on protected procedures. It could be a useful way to force reevaluation of barriers that depend on globals. That idea doesn't get much support. On the other hand, if we have protected interfaces, then we would allow **is null** (we would allow **is abstract** as well).

Erhard has comments on the wording. We go through his comments one-by-one:

3.4(3):... has [a]{one} parent type ...

3.4(25): The note needs clarification; it doesn't make sense. Steve Baird asks for guidance. Don't say "non-derived" rather talk specifically about interfaces. Don't use "this" in the sentence. Each note must be a self-contained story, since notes appear at end of clauses.

3.4.1(10) ... of [of] ...

3.9.4 (new section) First paragraph. The text "...thereby allowing multiple views of objects of the type." isn't the point of this feature. Drop this text.

The entire AI has too much text. Make it smaller, verify that all exa mples are correct syntax, etc. (in particular the discussion section, which only seems to get added to).

4.5.2(2): Must we introduce a new term here for "potentially share descendants"? Other alternatives are: weakly compatible, potentially convertible. Tucker doesn't think we need a term at all. Just use "convertible" and define the rules in 4.6. (That has to be done anyway.)

4.5.2(3): What is "tested type"? Steve Baird notes this belongs to membership operations. But how can we tell that? The entire wording section needs to be clearer on how it relates to the existing wording. Wording must identify whether it follows, precedes, replaces existing text and which paragraph(s).

8.3, after para 26:

The wording "both shall be overridden" is not liked; it sounds like you have to write two overridings. This is intended to handle the "diamond" inheritance pattern.

Again, the note must be self-contained.

Tucker suggests replace this by "If there are multiple implicitly declared homographs that are not overridden, they must be fully conformant."

Tucker wonders, can we allow a non-limited type to inherit from a limited interface? Why do we have this restriction? We need an example of a problem. Steve Baird replies that the compiler needs to know whether an object is returned by reference. One possible solution: not allow functions to return Limited_Interface'Class. We'd also have to have to disallow the same thing for Limited_Abstract'Class and indefinite formal limited private to avoid contract problems. AI-325 might resolve this? Or make it worse?

Approve intent of AI: 8-0-1.

AI-252/03: Object.Operation notation

Tucker explains the wording. Someone asks what happens if a component and an operation have the same name. A component always hides subprograms (direct visibility vs. use visibility). Subprograms overload each other (of course).

Erhard is concerned about class-wide operations. He finds it weird that an operation Foo can come from many places. Tuck explains that the choice between class-wide and dispatching is often an implementation detail, and one might put a class-wide wrapper around a dispatching operation, or change a class-wide operation to a dispatching operation to enable it to be overridden. We don't want to force users to change their calls in these circumstances.

Erhard expresses concern about case where class wide type is for a type declared in another package. It would be weird to declare such an operation. In any case, there is no ordering. The operations in all ancestor type's packages are considered, as in use-visibility. So the worst case would be an ambiguous call error.

Erhard suggests only considering class-wide operations where specific type is declared. Tucker says this would make the **use** clause users and the Obj.Op users see a different set of class-wide operations. It would be best if users could change calls from one form to the other without changing the routine called. (This would be a sort of Beaujolais effect.)

Move the last sentence of the new wording to dynamic semantics, as calls are a dynamic construct.

What about F.Op where F is a function returning an access type? You consider all the interpretations, including implicit dereference.

What about order of evaluation? 6.4 allows prefix and parameter evaluations to be in an arbitrary order, this does not change that.

Obj.Op is a name. Thus, you can rename it:

```
type T is tagged ...

procedure P (This : T; Blah : Q);

X :T;
Fun : Q;

X.P(Fun);

procedure Foo (Blah : Q) renames X.P;

Foo (Fun);
```

Jean-Pierre doesn't like this. It hides the purpose of the notation. Tucker points out that we can already do this with protected subprograms and all kinds of entries. And this could be useful in a generic formal, as it is for entries.

Pascal doesn't like the rule that this name has the Intrinsic convention.

Tuck explains that it is necessary to avoid requiring the implementation to generate a wrapper on 'Access or renaming-as-body. For instance:

```
type PT is access procedure (Blah : Q);

Z : PT := X.P'Access;  -- Illegal because the convention of X.P is Intrinsic.
```

Without this rule, we'd have to be able to generate a pointer to a call with partially bound parameters. That probably would require some sort of wrapper.

Why is this AI needed? It reduces a barrier to entry for programmers coming from other OOP languages. It also eliminates the difficulty of determining precisely which package an operation is declared in, which is especially painful for dispatching calls.

Jean-Pierre is concerned that this makes it harder to check that all subprograms are identified by a fully expanded name. In Ada 95, a search for **use** is sufficient to insure that, and that is very easy to find.

Randy points out that the inheritance of OOP breaks this anyway -- the routine is often not (explicitly) declared in the package to which the exp anded name refers. So this doesn't make it much harder to find the declaration (you have to search the same set of packages either way). Erhard says that this was even true in Ada 83 with derived type inheritance. Pascal suggests using an ASIS tool to check this (and other stylistic rules), not grep.

Jean-Pierre is not convinced. He fears that high-integrity users might be alienated. Alan points out that high-integrity users don't use tagged anyway — they can grep for **tagged** if they want (and thus avoid this new feature completely).

What about interfaces? Clearly, this needs to look for class-wide operations in interface packages. Luckily, the definition of ancestor includes those.

Jean-Pierre comments that if there are two controlling operands, this doesn't read too well. Sure, but you can always use a normal call in that case. Other OOP languages don't have the option of another format, and generally don't allow multiple controlling operands.

Approve intent of AI: 5-1-3. Jean-Pierre votes against, giving the reasons mentioned during the discussion.

AI-266-2/03: Task termination procedure

Alan explains the changes. Some confusion occurs: Alan sent a new version labeled "for those who missed it.", which was not filed as it appeared redundant. The editor sugge sts that new rewrites be clearly labeled as new.

Erhard asks what the default handler is for. It is for providing a partition wide-handler.

Tucker proposes the default only affect a task's children. He suggests the master task should be involved. Steve Michell says there is no such thing as master task; it makes more sense to use "master" (as in master block). Erhard doesn't think that master is the right grouping for allocated tasks. The creator may want to know what happens. Steve Michell says that can be handled by using explicit handlers.

Steve Michell worries about race conditions. Not a problem, because you only read these when you terminate. Jean-Pierre worries about the master walk to find the terminator. That doesn't seem to be a problem. Steve Michell thinks copying of handlers would be better. No, because that doesn't change the handler for tasks or masters that already exist – and that *would* cause a race condition.

If we adopt this semantics, then we should have separate routines for default handlers, because they are very different from a task-specific handler. But drop the default value for the task in the specific handler routines; it isn't useful.

Pascal and Tullio both note that Ada has a task hierarchy, we really ought to support it for these handlers.

Joyce wonders if it would be impossible to support handlers for library level tasks – they elaborate before any code can be executed. Tullio says that the static elaboration model (AI-265) could be used to handle that. Randy says that you could have a package that contained the setting of the default handler, and use pragma Elaborate_All on it to force it to elaborate before the package that creates the tasks.

Tucker summarizes the discussion so far: return to separate default procedures, eliminate the default parameters, default handlers are set for the current task and its children.

Erhard asks why the Old_Handler parameter is needed here. Do you really want to be able to replace the old handler? You might want to chain the handlers – canned libraries need their handlers executed, no matter what the rest of the application does.

What does old handler return? Does it include the default if there is no specific handler? No, and doing so would cause a problem. (If the default is changed after it was returned from setting a specific handler, the old handler would be called, not the current default.) But we do need a way to call the default handler if there was no specific handler. Tucker suggests we need "call default handler" routine, that would be called if there is no previous handler.

That is, chaining would look like:

```
if Old_Handler /= null then
      Old_Handler (...);
else
      Call_Default_Handler (...);
end if;
```

Tucker, thinking out loud, suggests that maybe we should have just one handler, with an enumeration status. That is safer, because then you can be sure that you didn't miss a case.

Tucker would prefer that the old handler return a deferred constant that represents the default handler if there is none. That way, chaining is just calling Old_Handler. That has some agreement.

Instructions for Alan: It is important that chaining work. And the task hierarchy should be used.

AI-285/04: Support for 16-bit and 32-bit characters

Pascal describes the current version of the AI.

A.4(1), A.6(1), A.7(13) are missing a comma before "and".

Kiyoshi will consult with the Japanese SC22 about the acceptability of this AI. He will provide input by e-mail prior to the next ARG meeting.

Pascal notes only "space" (32) character is considered a "space". A number of operations skip a space. E.g Get in Text_IO, Trim in the string packages, etc. Should this be changed to recognize ideographic space? But this is not upward compatible. Kiyoshi's view is that space (32) is the only space. Should non-breaking space at position 160 be treated as a space? Probably not, both because of upward compatibility and because it is not supposed to be a separator.

Pascal wonders what C is doing for 32-bit characters? Should we add something to Interfaces.C? Pascal will investigate.

Jean-Pierre is concerned about the tables for character classification and upper/lower case equivalence. Pascal has already created those from the official character set definitions; see the appendix of the AI.

It is noted that we don't have much choice on this issue; this is an SC22 mandate. Are there any other SC22 mandates? There is a standard for writing SC22 standards. That standard tries to make the terminology for programming languages consistent. We can't practically use that standard until we do a full revision; otherwise the terminology would be hopelessly confused. We of course are following the character set mandate.

Approve intent of AI: 9-0-1.

AI-292/00: Sockets Operations

Randy reported that he has been unable to find people willing to write this up (beyond himself, which the ARG specifically asked him not to do). There is some sentiment (which Randy does not share) that sockets shouldn't be included in the standard.

Tucker asked to consider using GNAT.Sockets. Randy noted that there were a number of concerns with those packages, not the least of which is that no one has stepped forward to write an RM description of it.

After much discussion, Ed agreed to see whether resources could be found to document GNAT.Sockets to RM standards.

AI-296/02: Vector and matrix operations

John describes the changes and work for this package. Vectors and matrices are pretty obvious, but linear algebra makes it more interesting. He visited NAG (National Algorithms Group) to try to get ideas. Everything of theirs has lots of parameters, giving a lot of control but also making it fairly hard to use. They also make the LU decomposition visible so user can do it if needed.

Possible additions. Finding roots is a possible addition. To find more, John looked at his HP Pocket calculator. If the calculator can do it, it seems that Ada should be able to. The calculator has:

- Trig on real and complex.
- Does not have eigenvalues.
- Statistical stuff.
- Integration for function in 1 variable.

Ed comments that vectors and matrices are fundamental, but statistics seems like it is not appropriate to a language standard.

John notes that he included a vector product (aka "cross product"), but perhaps that was a mistake. It only makes sense for 3-D vectors. It is suggested that a subtype Three_D_Vector be defined to make this clearer.

Steve Baird thinks that Solve should be called "/". John replies that that may confuse the user because order of operations is not what you expect. Solve solves a set of equations Y = solve(A, X) so that X approx = A * Y. Moreover, it could cause people to inverse the entire matrix when that's not necessary.

John notes that the package includes operations mixing real and complex operations for complex vectors and matrices.

John asks if there should be a preinstantiated generic, all language-defined generics have one (also see AI-328 below). Yes, and it should be described similarly to G.1.2(9/1).

Pascal notes that there are no operations involving "pure" imaginary operations. That is intentional.

Add LU decomposition as separate operation? Yes, this makes a big difference in performance in some applications (the LU decomposition does not need to be recalculated each time.) There should be a separate type (private, array?) for LU decompositions.

John: I placed the eigenvalues and eigenvectors in child packages, so they would be easy to remove. There is an issue: a square symmetric real matrix has real eigenvalues. Otherwise, the matrix can have complex eigenvalues. Thus, the package has 4 routines:

- Symmetric real matrix for real values;
- Nonsymmetric real matrix with complex eigenvalue;
- Hermitian matrix;
- General complex matrix.

These routines all have implementation-defined accuracy requirements.

Who is the audience for these packages? A typical engineer.

How do we check whether a matrix is symmetric? We use floating point equality, and raise Argument_Error if not exactly symmetric.

We decide to remove the cross product.

Integration is not worth it, because there are so many algorithms, all tuned to particular kinds of data. That is too complex for our purposes. (The NAG Integration package runs 27 pages.)

Should we include routines for finding roots of polynomials? John will try to do something for this. After all, the NAG Polynomial package *only* has 12 pages.

Approve intent of AI: 7-0-3.

AI-297/05: Timing Events

The !proposal doesn't match the !wording; that is very confusing. It doesn't add anything anyway, just replace it with "See wording".

Pascal wonders what a "potentially suspending operation" is. There is no such thing. Use "potentially blocking operation".

Steve Michell thinks that the parameters "In_Time" and "At_Time" should have the same name. No, these parameters have different meanings (relative vs. absolute time).

Jean-Pierre asks why we are doing a ceiling check; ceiling priorities may not apply to this unit. Alan says that this is a copy of the text from interrupt handlers. No, that text only applies when the ceiling policy is in effect. Moreover, this wording assumes that timing happens at the highest priority; that is over-specification. Tucker points out that by specifying this to have the highest priority, we make it safe to call in all cases. Otherwise, it would be possible that the clock handler couldn't call this procedure without causing a bounded error. Alternatively, we'd have to have a constant with the clock priority, so that could be used in pragma Ceiling_Priority.

Jean-Pierre worries that if you've missed a lot of deadlines, you could get a stack overflow because the rearming causes an immediate call. (That is, call handler, which rearms, which calls handler immediately, which rearms, etc.) There doesn't seem to be a reasonable solution to this, and if you've missed a lot of deadlines, you're in big trouble anyway.

To summarize the changes: Replace proposal with (see wording.); "potentially suspending operation" –> "potentially blocking operation"; add "if ceiling_locking policy applies…" to the ceiling check.

Approve AI with changes: 10-0-3.

AI-298/04: Non-Preemptive Dispatching

This AI is removed from "hold" state, since AI-321 (on which is depends) has been approved.

AI-307/03: Execution-time Clocks

Alan explains the changes - see the first paragraph of the discussion for a summary.

John sees typos marked in his notes. He withdraws the first after discussion. The second is the second paragraph of Initialize and Finalize - "Ada.Finalize.Controlled" should be changed to "Ada.Finalization.Controlled".

Randy suggests eliminating the first paragraph of the discussion, because we don't talk about differences in AIs (that can be found via the version control system if needed).

Tucker notes typos in the first sentence of Implementation Requirements: "user" -> "used"; "must" -> "shall".

In the last paragraph of !discussion – change "ARM" to "Arm".

Pascal opines that the discussion is long and boring. Some should go to !problem, some should stay where it is, and the rest cleaned.

Steve Michell comments that the phrase "the timer associated with the task" implies that there is only one timer, but there is nothing preventing multiple timers for a task. Yes, there can be multiple timers, but there can be only one clock, so this is OK.

Erhard can't find where Timer_Resource_Error is raised. That's in the 6th paragraph.

Pascal thought that the previous discussion concluded that Timer_Resource_Error wasn't valuable. No, the discussion was inconclusive. Alan comments that the prototype developers thought this was important, so he left it in.

Tucker: "When Timer is created…" sounds like that it has something to do with object creation. Say "For a given Timer, at a time no later than return from the first call of one of its Arm procedures, the …". This later is modified to " When a Timer object is created, or upon the first call to one of its Arm procedures, ..."

[At this point on Friday afternoon, the editor's computer failed. Tucker Taft volunteered to be the emergency minute-taker. The editor was able to recover his notes after returning home.]

Comments about exceptions being raised should be removed from the package specification.

Remove "In this Annex," from the first (text) paragraph of Static Semantics. The last sentence of that paragraph should be changed to "It is implementation defined which task, if any, is charged the execution time..."

Add "values of" to the second sentence of the second paragraph of Static Semantics: "The set of {values of} the type CPU_Time corresponds...".

In the description of the Arm that takes a CPU_Time parameter, change "counts" to "monitors".

Does the Timer_Expired description cover all of the possibilities? We draw a state diagram on a whiteboard. The wording needs adjustment. Rewrite the Timer_Expired paragraph to "The Timer.Timer_Expired protected entry suspends the calling task until the timer expires if the timer is in the armed state; if the timer has already expired, then the calling task proceeds. If the timer is in the disarmed state, the Timer_Error exception is raised."

Throughout: remove [Timer.], as it is just clutter

Approve AI with changes: 12-0-1.

AI-321/03: Definition of dispatching policies

Alan explains the changes to the wording. It was restructured as discussed at the last meeting. But ignore the !proposal section; it wasn't updated.

Last three paragraphs of the new D.2.2 should move. Note 14 and the preceding paragraph should be at the end of D.2.1; Note 15 should be at the end of D.2.3.

The cross reference (see 9.2), which appears in D.2 and D.2.1 should be (see 9) -- the reference is to 9(10).

Tucker comments that the sentence "A task runs (that is, it becomes a running task) only when it is ready (see 9)" in D.2.1 feels pre-emptive. John suggests changing "only when it is ready" to "only if it is ready". Erhard suggests "A task can become a running task only if it is ready (see 9) and the execution…"

Steve Baird asks about the first paragraph of the Documentation Requirements in D.2.3. Shouldn't "the highest priority" be a "a higher priority". Joyce suggests fixing with "the highest priority of those available". Alan suggests that the wording should be "nonempty ready queue". That gets general agreement.

Jean-Pierre asks about the last paragraph of D.2.1 Dynamic Semantics. He describes an OS that gives slices based on priorities (bigger slices for higher priorities). He doesn't think that this paragraph would allow that (where lower priority tasks may run). After some discussion we decide that there is no problem here.

Erhard wonders why we select from the head, but never explain what the head is. "Head" is what you select; you could define "head" any way you want for a specific policy.

John notes a typo: "possible" should be "possibly" at the end of D.2.1 Dynamic Semantics. He also suggests fixing D.2.2 Implementation Permissions by removing the word "other". Randy notes that we already fixed that text in AI-256; we should use that fix here.

Approve AI with changes: 13-0-0.

AI-326/01: Tagged incomplete types

Can a dereference having an incomplete type be passed to a subprogram with a formal that is of an incomplete-tagged type? Tucker thinks that you should be able to do what you can do with access values and access parameters. Randy and Steve Baird object; you must not allow accesses to a type that is not yet frozen, as the position of the components has not yet been determined. That's clearly a problem for discriminants and for tags (in implementations that allow the tag to be positioned). Ada 83 had this problem, and we certainly don't want to reintroduce it.

Tucker wonders if this problem already exists for children that can see the incomplete type and try to call a dispatching operation. Perhaps, but clearly an implicit dereference is occurring to extract the tag when you use a controlling access parameter, and such an implicit dereference is illegal. So we can apply Dewar's rule here: the language certainly does not say something absurd (that such calls should work).

Tucker decides to withdraw the dereference rule. Thus, we decide to drop the "or actual parameter" from the summary, problem description, and proposal.

Can a primitive of an incomplete tagged type be declared before the full type? Yes, you can do that. That is already true if you use access parameters on a "normal" incomplete type. But such an operation is not a dispatching operation.

Approve AI with changes: 10-0-1.

AI-327/01: Dynamic ceiling priorities

The only primitive needed is to be able to set the ceiling priority from inside the protected object. A change takes effect at end of the protected action. The idea applies to *all* protected types, attributes should be used to avoid having to define a notion of predefined protected-object identifiers.

The pragma should be a restriction: No_Dynamic_Ceilings.

Jean-Pierre Rosen suggests an attribute available from outside of the protected object. Changing the priority from outside brings up a number of problems (covered in the discussion section); it is a can of worms.

What value does 'Get_Ceiling return after a call to 'Set_Ceiling prior to returning from the current protected action? We don't want to have to keep multiple versions of this value (the "real" one and the "about to be used" one).

Suggestion: Make this into an attribute that represents a variable component of the protected object, that is only accessible inside the protected body. (The wording would be similar to that for 'Caller.)

Call the attribute 'Priority to match the name of the pragma Priority, since it is independent of the ceiling locking policy.

Straw vote to keep AI alive: 12-0-2.

AI-328/01: Add preinstantiations of Complex_IO

Fix spelling in penultimate sentence: corr{e}sponding.

Approve AI with change: 9-0-0.

Detailed Review of Regular AIs

AI-237/05: Finalization of task attributes

Erhard explains his objections to the current wording. The basic problem is that finalization immediately after task termination is only mentioned in an imple mentation permission, rather than being part of the "main" semantics.

We discuss the issues extensively. (We also discussed this in Bloomington.) We eventually settle on the following wording:

"After a task terminates, all of its task attributes are finalized. At the time when the master of an instantiation of Ada.Task_Attributes is finalized, any remaining non-finalized attributes associated with the instantiation are finalized.

Implementation Advice:

Finalization of task attributes and reclamation of associated memory should be performed as soon as possible after task termination."