Minutes of the 44th ARG Meeting

24-26 June 2011

Edinburgh, Scotland

Attendees: Steve Baird, John Barnes, Randy Brukardt, Alan Burns (Friday only), Jeff Cousins, Gary Dismukes, Bob Duff, Brad Moore, Erhard Ploedereder, Jean-Pierre Rosen, Ed Schonberg, Tucker Taft, Tullio Vardanega (except Sunday).

Observers: None.

Meeting Summary

The meeting convened on 24 June 2011 at 12:30 hours and adjourned at 12:20 hours on 26 June 2011. The meeting was held in St Leonard's Hall on the Edinburgh First conference site. The meeting covered the entire agenda.

AI Summary

The following AIs were approved with editorial changes:

AI05-0249-1/01 AI05-0130-1 should be a pathology (7-1-4)
AI05-0250-1/01 Thoughts on type invariants (10-0-1)
AI05-0251-1/01 Problems with queue containers (11-0-1)
AI05-0252-1/01 Questions on subpools (6-0-5)
AI05-0253-1/01 Accessibility of allocators for anonymous access of an object (10-0-2)
AI05-0254-1/01 Do we really have contracts right? (9-0-3)
AI05-0255-1/01 Missing text about user-defined iterators (13-0-0)
AI05-0257-1/01 Insert returning a Position (12-0-0)
AI05-0258-1/01 Preserving information in an exception occurrence (12-0-0)
AI05-0259-1/01 Default convention of anonymous access-to-subprogram parameters (12-0-0)
AI05-0260-1/01 Pedantic question on the wording of the definition of modulo (11-1-0)
AI05-0263-1/01 No_Dependence should sometimes be excluded from applying to the run-time (10-1-0)

The following AIs were discussed and a resolution was postponed until after the Ada 2012 revision is completed:

AI05-0012-1/02 Independence and Representation clauses for atomic objects
AI05-0256-1/01 RCI units should not allow types with user-defined stream attributes
AI05-0261-1/01 Default storage pools for storage pools

Detailed Minutes

Previous Meeting Minutes

The previous minutes were approved during the April phone meeting.

Date and Venue of the Next Meeting

Next meeting is associated with the SIGAda conference in Denver, November 11-13, 2011 (WG 9 is the afternoon of the 10th). After that, the next meeting will be in Stockholm, June 15-17, 2012, immediately after the Ada-Europe conference. We shouldn't need a February meeting this year.

RM Review Plan

Randy asks when the best time to have the final ARG review of the draft standard. The group agrees that after Draft 13 is ready is best, presuming that this is not too long after the meeting.

WG 9 has changed this from an Amendment to a Revision, so we don't need to review or edit the Amendment document.

At the close of the meeting, we review the review plan. The review should close about Sept. 1, so it can get to WG 9 as promised. Try to start as soon as possible; minor editorial things could be left out.

Thanks

Thanks to Randy for his tireless efforts as editor -- and tell him to hurry up with the draft standard.

Thanks to the host for the fine accommodations.

Old Action Items

ASIS assignments are still open for most people. Randy did not quite finish the draft Standard. All other items are now closed.

New Action Items

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

Everyone:

- Complete their ASIS assignments as previously sent by Jean-Pierre Rosen.

Randy Brukardt:

- Complete the draft Standard.

Editorial changes only:

- AI05-0250-1
- AI05-0251-1
- AI05-0252-1
- AI05-0253-1
- AI05-0254-1
- AI05-0255-1
- AI05-0257-1
- AI05-0258-1
- AI05-0259-1
- AI05-0263-1

Jean-Pierre Rosen:

- Create an !ASIS section for AI05-0188-1 (see October minutes)

Detailed Review

The minutes for the detailed review of AIs and SIs are divided into ASIS Issues (SIs) and Ada 2005 AIs (no SIs were considered at this meeting). The AIs and SIs 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 had six votes for, one vote against, and two abstentions.

Several times during this meeting, the ARG split into subgroups to create specific wording proposals; these proposals were later presented to the entire group for approval. Only the results of these subgroups is recorded here; details of their work is not recorded.

If a paragraph number is identified as coming from the Amendment (with a /2 version), the number refers to the text in the Final Consolidated AARM. Paragraph numbers in earlier drafts may vary.

Detailed Review of Ada 2005 AIs

AI05-0012-1/02 Independence and Representation clauses for atomic objects

Randy notes that he put this AI on the agenda because it doesn't seem insolvable and the question has some importance. Last time (Albuquerque), we got stuck on the meaning of Pack.

Erhard says he thinks that it is important that the problem of the AI is solved.

Tucker says that it will take more time than we have left in the meeting to come to an agreement on this AI, he still opposes the approach taken in the AI. [Editor's note: this was discussed with about 60 minutes left in the meeting.]

We decide to continue to defer this AI.

AI05-0243-1/05 Clarification of categorizations

John had complained at WG 9 about having both pragmas/aspects for packages, and asked for reconsideration. He does not want to have both aspects and pragma. The concern is program maintenance, having both (especially in the same program) can be confusing.

Tucker notes that there are two possible solutions here: one is to make the pragmas (Pure, Preelaborate, and so on) obsolescent, and other is to drop the aspects.

Most of us prefer using the aspect and making the pragma obsolescent (such that it is not taught at all). Dropping the aspects is not the way we want to go.

Randy notes that the original reasons for retaining the pragmas was a dislike (by John and a few others) of the aspect syntax for packages, and because we were concerned about the number of changes needed to replace all of the pragmas in the predefined packages. The latter issue seems moot – we have decided to change a similar number of pragmas to obsolesce pragma Convention (mostly for convention Intrinsic). So if having both is not enough to assuage John's concerns (John was not at the February meeting where this was discussed), we might as well obsolesce the pragmas and thus make the language more consistent. Someone like John who really doesn't like the aspects can still use the pragmas.

John still prefers the pragmas and no aspects, he really, really does not want the pragma to go away (even logically). He realizes, however, that his request for reconsideration is likely to have the opposite effect to the one he intended. So he withdraws his request for reconsideration.

AI05-0249-1/01 AI05-0130-1 should be a pathology

Real code probably exists, but it is hard to believe that it matters sufficiently to force compilers into an incompatibility (with previous versions of themselves).

Approve AI: 7-1-4.

Bob Duff says he has no explanation for his no vote.

AI05-0250-1/01 Thoughts on type invariants

(1) Steve had sent wording to allow use of invariants on the private part.

There is some question as to whether the invariants really need to be in the visible part in the first place. It seems to weird to disallow it here; the information might be useful to the caller. But there is no effect on the user – they should not be able to change the value themselves (it's something that the implementation guarantees).

Steve had sent wording for this change (June 20th, after the agenda was sent out); we'll use his wording. The "which" police (Bob) says that "which" should be "that" in this wording.

Tucker argues that we don't need Type_Invariant'Class at all, since it doesn't have any effect on the caller. Bob worries about children that are inheriting routines from their parent. Erhard worries that it should apply all the way down the tree.

Tucker suggests that Type_Invariant'Class be required to be visible (since it is inherited). The programmer creating an extension needs to know about it. Type_Invariant is not inherited and can be in the private part.

Class-wide invariants would need to be dispatching calls. Bob says that the AdaCore customer wanted to write invariants that look at the components directly (which requires being in the private part).

Ed suggests that we should simply allow all of this; we don't have a enough knowledge to determine how these will be used, and restrictions seem to be mostly methodological. [Editor's note: I don't see how the need to know about such an invariant in extensions is "methodological".]

We have similar restrictions for preconditions and postconditions, and we even added expression functions to allow the functionality of looking at components of private parts in a visible expression. Type invariants aren't different. So there is no functional requirement to allow these in the private part, and it only makes sense to allow them there if clients can't be affected by them. That's true for specific invariants and not true for class-wide invariants.

At this point, Erhard asks about an unrelated subject, and we never went back and took a vote on the solution to question (1).

Erhard is worried that view conversions of a private child to a parent type, which is passed to a parent operation which can break everything that the child type requires. Tucker says that these conversions are very evil, and Ada should never have allowed them.

Tucker suggests disallowing evil conversions when invariants are involved. A private child with invariants cannot be converted to a private parent. The problem is a static call on a parent operation. "Can't do a writable view conversion if the type has an invariant." Steve says that would be a generic contract problem; Jean-Pierre says that would be privacy breaking (with the extension supported by question (1)).

Tucker wonders if we could require a invariant check after such a conversion. Steve ask what happens if you have a rename; you would have to make the check when the renames goes away.

Erhard suggests associating invariants with the object rather than the type. Tucker says in that case, "a view conversion still requires checking the invariants of the original object". The invariant of the object is not destroyed by any means. That seems like the better model.

So invariants belong to objects, not to operations. Effectively, these are always dispatching.

We'll let one of the subgroups work on wording of this off-line.

On question (3), we agree that we have "internal" and "external" right, at least until the subgroup screws it up. (This was extensively discussed in e-mail before the meeting.) So we don't need any change for that.

On question (2), this scheme seems premature; we need experience to figure out what we need. Implementation and user experience are both needed here. The proposal would be something similar to GNAT pragma Pure_Function, but that is an assertion rather than a requirement.

Steve brings up the memo function case. It quickly becomes clear we don't have any sort of consensus, so we will not do this.

On Saturday, we got a report from the wording subgroup. It got totally bogged down in trying to identify and patch additional holes – and didn't produce any wording.

Erhard had sent some thoughts on this topic to the ARG mailing list.

Jean-Pierre shows the canonical example that was causing problems:

```
type Ordered_Tree is tagged ...

type Balanced_Tree is new Ordered_Tree with ...;

procedure Insert (T : in out Balanced_Tree) is
begin
    Insert (Ordered_Tree(T));
    -- We want the invariant to be Ordered_Tree, the tree is not balanced here.
    -- Balance the tree.
end Insert;
```

A client doing the same:

```
V : Balanced_Tree;

Insert(Ordered_Tree(V)); -- Ought to fail if not balanced.
```

Tucker notes that this latter case should have been illegal; but it's too incompatible to do that now. He says that a restriction could (and should) be defined to make this illegal. If there is an intervening private type (including the starting subtype), then the conversion is illegal.

Tucker thinks that all we need to do is make a check on the view conversion upon return (or assignment) at the end of the use.

Steve wonders if we really need to do anything; visible components are already punted on (if a visible component is changed, there is no requirement to recheck invariants). What's one more hole? Tucker thinks that this is impossible for designers of packages to prevent (the package designer can't prevent derivations or conversions to the package's type), while it could screw up the innards of the package.

Bob notes that other loopholes are under the control of the package that declares the type. (To prevent problems with visible components, just don't declare any.) That's not true of this loophole. That argues strongly that it needs to be plugged.

Tucker wonders if access parameters designating T should be covered. Steve says that array of T also would need to be covered. What about other components? It seems like it needs to be "a part of T".

Steve worries about the performance hit about checking a giant array, if the operation just changed one item. That seems unavoidable, we don't want to be able to avoid the checks by just converting to an array of length one. The programmer can avoid such a check by using a temporary object if necessary.

Erhard suggests only allowing type invariants on “properly encapsulated” types; they could not have any exported routines that that have any higher level types (such as array of T). That seems pretty limiting (Randy notes that Claw has a number of such routines).

Tucker enumerates choices (for the invariant on type T):

(1) Drop type_invariants

(2) Check all parts of T (including access-to-T)

(3) Disallow exporting ops that manipulate parts of T.

(3A) Disallow composites with parts of T; access-to-T is OK.

(4) Disallow exporting types with parts of T.

(5) Present proposal.

For (2), Bob wonders if all access-to-T in parts of are covered. That would require walking all attached nodes in a data structure, and that isn't possible (the compiler can't know the data structures involved). So where is the line drawn? Tucker suggests that only the “exposed” parts are checked.

So which of these are acceptable?

We're surely not going to do (1) because of this problem alone.

(3) seems rather goofy and limiting. (3A) and (4) are just variants of (3), and they're no better.

So that leaves (2), it doesn't sound great but it sure is a lot better than any of the alternatives. We'll clearly need to pass this off to a wording subgroup.

Erhard asks about exceptions raised by a call. These do not check after the call in that case. The wording probably is unclear; Randy thinks that invariants (and postconditions) are checked only after “normal completion” of the subprogram. The postcondition wording already says "after successful return from a call", so we probably ought to use that.

Erhard thinks that checks should be done on the way in. Bob points out that doing checks on the invariant would be a problem when the invariant calls a function of the type (which is the normal case) – you'd have infinite recursion.

Bob thinks that checks on the way in could be an assertion mode, but the default should be what the standard currently says to avoid the infinite recursion problem.

After a wording subgroup breakout session, the group sent a redo of AI05-0146-1, which is an obsolete version of the wording.

They added a “redundant” rule about “applies” for regular invariants in front of 7.3.2(7/3).

The word “successful” was added to most of the existing bullets of 7.3.2(9-16/3).

The third bullet is new:

- After assigning to a view conversion, outside the immediate scope of T, that converts from T or one or more record extensions (and no private extensions) of T to an ancestor of type T, a check is performed on the part of the object that is of type T; similarly, for passing a view conversion as an **out** or **in out** parameter outside the immediate scope of T, this check is performed upon successful return;

Erhard wonders if the conversion that comes from inherited code triggers this rule. It seems like it should.

Steve asks whether the checks that are needed are known statically. He wonders about a dereference that designates a view conversion. We're not trying to deal with such cases, we know that there are some holes.

The fifth bullet is rewritten:

- Upon successful return from a call on any subprogram or entry that:
  - is explicitly declared within the immediate scope of type T (or is a part of an instance of a generic unit declared within the immediate scope of type T), and
  - is visible outside the immediate scope of type T or overrides an operation that is visible outside the immediate scope of T,
  - has a result with a part of type T, or one or more IN OUT or OUT parameters with a part of type T, or an access to variable parameter whose designated type has a part of type T.

  the check is performed on each such part of type T.

Should be “and” at the end of the second subbullet.

Steve is writing an AARM note to say that the checks needed can be determined at compile-time.

Steve thinks the text “(or is a part of an instance of a generic unit declared within the immediate scope of type T)” is ambiguous.

Replace with: (or is within an instance of a generic unit, and the generic is declared within the immediate scope of type T)”

“within” is wrong. Randy says that “part” is not defined this way.

Erhard suggests “or by an instance of a generic unit, and the generic is declared within the immediate scope of type T)”. That's sounds good.

7.3.2(18/3) needs to steal wording from previous work. Another subgroup task.

Bullet 3: “After assigning, outside of the immediate scope of T, to a view conversion that converts from T...” That isn't better, leave this bullet alone.

Approve this part of this AI: 8-0-4.

After another subgroup wording session, we again turn to this AI.

Erhard says that invariants get checked by the implicit view conversions that are part of inheritance (and dispatching). So it appears we don't need any more rules here; 7.3.2(18/3) plus the newly added check says everything we need.

Randy notes that we aren't allowing Type_Invariant'Class on interfaces; there is no such thing as a private interface. Tucker wonders if we want this; all you can do in it is call a dispatching routine. That does not seem very useful, especially as these are mainly intended for the class implementor.

So we'll leave the rules such that invariants can't be used on interfaces; having done that, we don't have the problem that 7.3.2(18/3) was trying to fix, so we don't need the second sentence of it at all. [Editor's note: the original notes said that the entire paragraph of 7.3.2(18/3) was to be deleted, but that is wrong – we only want to delete the part added by AI05-0247-1 – the first sentence is necessary to explain the semantics of access-to-subprogram routines at a minimum.]

Tucker says we need to modify 3.9.2(20.4):

> otherwise, the action is the same as the action for the corresponding operation of the parent type or progenitor type from which the operation was inherited {(except that additional invariant checks – see 7.3.2 – and class-wide postcondition checks – see 6.1.1 – may apply)}. If there is more than one such corresponding operation, the action is that for the operation that is not a null procedure, if any; otherwise, the action is that of an arbitrary one of the operations.

Erhard suggests eliminating the outer parens:

> otherwise, the action is the same as the action for the corresponding operation of the parent type or progenitor type from which the operation was inherited {except that additional invariant checks (see 7.3.2) and class-wide postcondition checks (see 6.1.1) may apply}. If there is more than one such corresponding operation, the action is that for the operation that is not a null procedure, if any; otherwise, the action is that of an arbitrary one of the operations.

Erhard will write a note explaining more.

Approve AI with (massive) changes: 10-0-1.

Erhard read his note:

> Note: A call of a primitive subprogram of type NT that is inherited from type T will need to satisfy the specific invariants of both the types NT and T. A call of a primitive subprogram of type NT that is redefined for type NT need to satisfy only the specific invariants of type NT.

13.3.1(17-28/3) define when an aspect “applies” to an entity; there should be some italics and index entries here. 7.3.2 is using those rules, so it is important that it is in the index.

AI05-0251-1/01 Problems with queue containers

Question (1): The indefinite queues are essentially useless as Dequeuing requires predicting the tag and discriminant values of the value to be Dequeued. It is possible to instantiate a definite queue with a holder container, which is much more useful. So eliminate the indefinite queues.

We need a user note to explain how to do this (otherwise the lack of indefinite queues will look like a mistake).

Question (2): Tucker had suggested making the priority an entry family. The priority then has to be discrete.

Randy had objected that time is a common priority and it cannot easily be mapped to discrete.

How could this routine be implemented? Requeue is needed. Alan explains that you would need to use two additional queues so that the order of calls is not scrambled. (First requeue on one; second requeue on the other, then back to the first). While this is not an obvious implementation technique, it is possible.

Tucker suggests replacing this with a polling procedure. Such a routine would either return a high-priority object or an indication of failure; the caller would have to poll the routine in that case.

```
not overriding
procedure Dequeue_Only_High_Priority
  (Low_Priority : in     Queue_Priority;
   Element      :    out Queue_Interfaces.Element_Type;
   Success      :    out Boolean);
```

Erhard is confused about the ceiling priority and the queue priorities. These are different things and are not related. There is a suggestion to call these Ordered queues. This idea is not well-liked; "priority queues" is a well-known concept and we don't really want to invent new ones.

Someone complains that Synchronized_Queues and Ordered_Queues seem to imply that the latter is not synchronized. Synchronized_Ordered_Queues is just too long.

Jean-Pierre says that he would like to be convinced that the blocking vs. non-blocking semantics is better.

Tucker says that we have been having problems coming up with a realistic case that uses the blocking one, but for the polling version we have well-defined, common cases.

Question (3) will be taken care of naturally by the rewording to handle question (2).

The wording details will be handled later in a subgroup.

On Saturday, after a subgroup meeting, we consider the wording proposed by the subgroup.

```
not overriding
procedure Dequeue_Only_High_Priority
  (Higher_Than : in Queue_Priority;
   Element : in out Queue_Interfaces.Element_Type;
   Success : out Boolean);.
```

If the queue is non-empty and the element *E* at the end of the queue satisfies Before (Get_Priority (*E*), Higher_Than) then *E* is removed from the queue and assigned to Element, and Success is set to True; otherwise Success is set to False and Element is unchanged.

Note: Unlike other language-defined containers, there are no queues whose element types are indefinite. Indefinite types can be catered for by using a holder container (see A.18.18) or by using explicit access types.

Jean-Pierre wants a clarification: “removed from the {head of the} queue”

Tucker would prefer the parameter should be tested as “>=”. The condition should be “not Before (Higher_Than, Get_Priority (E))”.

The formal could be called “Not_Before”. The other suggestion be “Minimum_Priority”. Tucker says that assumes that the ordering is lowest to highest. So we should use “Not_Before”.

Turning to the note, “catered for” is hated by Gary and Randy.

Randy suggests replacing the second sentence by “Elements of indefinite types can be handled by using a holder container (see A.18.18) as the element of the queue, or by using explicit access types as the element.”

Add an AARM note to explain why they are not there. (Can't have entries that are functions; using a out parameter would require guessing the constraints and tags of the result, which doesn't make any sense.)

Add an AARM Note to mention that Dequeue_Only... is not blocking.

Tucker announces that “Not_Before” is wrong, he suggests a number of options from which we select “At_Least”.

Tucker sends the following:

```
not overriding
procedure Dequeue_Only_High_Priority
  (At_Least : in     Queue_Priority;
   Element  : in out Queue_Interfaces.Element_Type;
   Success  :    out Boolean);
```

If the queue is non-empty and the function Before(At_Least, Get_Priority(*E*))
returns False, where *E* is the head of the queue, then *E* is removed from the queue and assigned to Element, and Success is set to True; otherwise Success is set to False and Element is unchanged.

Ed suggests:

If the head of the non-empty queue is *E*, and the function Before(At_Least, Get_Priority(*E*)) returns False, then *E* is removed from the queue and assigned to Element, and Success is set to True; otherwise Success is set to False and Element is unchanged.

There is a suggestion to improve the note by trying to combine “as the element”, putting it first. The editor will work that out. [His version follows:]

Note: Unlike other language-defined containers, there are no queues whose element types are indefinite. Elements of an indefinite type can be handled by defining the element of the queue to be a holder container (see A.18.18) of the indefinite type, or to be an explicit access type that designates the indefinite type.

Approve AI with changes: 11-0-1.

AI05-0252-1/01 Questions on subpools

We start with question (1). Tucker asks how a pool supporting subpools is intended to be used by the client (the pool implementor's view is more complex, of course). Randy gives the following outline:

- A pool object is specified for an access type (explicitly or via pragma Default_Storage_Pool);
- Call Create_Subpool (or an equivalent) to get a handle;

- Do a bunch of allocations that use the subpool (by specifying the handle on **new**).
- Call Unchecked_Deallocate_Subpool to free the contents of the subpool. This nulls the subpool handle. The client should not use the handle afterwards.

The pool implementor may reuse subpool objects (and thus the handles), but the client cannot reuse handles.

The only things the client uses in a pool supporting subpools package is the Create_Subpool routine and type Subpool_Handle. (along with allocators, and Unchecked_Deallocate_Subpool). Everything else is only for the pool implementor or Ada implementations use.

Could this be made clearer somehow? Randy doesn't think so, there are too many interrelationships. We'll leave this question for one of the subgroups to consider.

We think this model is fine, so we're not going to make any change to it. Thus, there will be no change related to question (1).

For question (2), no one has any comments on the new text. We'll let the reviewers check it more carefully.

Question (3) seems to indicate a real problem. It seems that some sort of run-time check is required; we'll let one of the subgroups work on that.

For improving the readability of the package, the subgroup suggests adding a comment line after Create_Subpool (13.11.4(7/3)) to separate the client part of this package from the pool implementor part, along with the description: “The following operations are intended for pool implementors:”. That seems like a good idea.

To fix the problem of question (3), the subgroup recommends the following changes:

> Add a Legality Rule after 13.11.4(22/3):
>
> For a formal parameter of Root_Storage_Pool'Class, the actual object shall not be a storage pool that supports subpools. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.
>
> Add a Dynamic Semantics rule after 13.11.4(23/3):
>
> For a formal parameter of Root_Storage_Pool'Class, a check is made that the tag of the actual object does not identify a type that supports subpools. Program_Error is raised if this check fails.

Tucker thinks this rule is far too pervasive. Randy says that we have to catch any case where the accessibility level is counterfeited (such as parameter passing). Bob noted a similar problem with access types, thus this rule.

Tucker claims that we always have the ability to counterfeit with access types, so we would have to ban those altogether. That way lies madness.

Tucker suggests just checking the rule whenever a storage pool is specified. But you don't know anything about the accessibility level there, so all you can do it fail everything. That seems way too limiting.

Someone notes that pool-specific access types don't have this problem of counterfeiting the accessibility level. It's also true that a library-level general access type has no problem (everything there has to be library level). So this isn't as mad as it originally seemed.

Given a storage pool: `for T'Storage_Pool use Obj;,` we need wording like "if Obj is descendant of Root_Storage_Pool_with_Subpools'Class then the Obj shall not be a formal parameter or a dereference of an general access value that is not library level." But we need both static and dynamic versions of this rule.

This is going to require another subgroup to write wording.

The second subgroup provides the following wording on Sunday morning:

> 13.11.4(22) says:

> The accessibility level of a subpool access type shall not be statically deeper than that of the storage pool object.
>
> Add here:
>
> If the specified storage pool object is a storage pool that supports subpools, then the name that denotes the object shall not denote part of a formal parameter, nor shall it denote part of a dereference of a value of a non-library level general access type.
>
> 13.11.4(23) says:
>
> When a subpool access type is frozen (see 13.14), a check is made that the accessibility level of the subpool access type is not deeper than that of the storage pool object. Program_Error is raised if this check fails.
>
> Add after 13.11(19):
>
> When an access type with a specified storage pool is frozen (see 13.14), if the name used to specify the storage pool object denotes part of a formal parameter or part of a dereference of a value of a non-library-level general access type, then a check is made that the tag of the storage pool object does not identify a storage pool that supports subpools (see 13.11.4). Program_Error is raised if this check fails.
>
> AARM Note: See 13.11.4 for an explanation of the restriction against use of formal parameters and general access value dereferences.
>
> 13.11.4(23a) says:
>
> Reason: This check (and its static counterpart) ensures that the type of the allocated objects exist at least as long as the storage pool object, so that the subpools are finalized (which finalizes any remaining allocated objects) before the type of the objects ceases to exist. The access type itself (and the associated collection) will cease to exist before the storage pool ceases to exist.
>
> Add after that:
>
> We also disallow the use of formal parameters and dereferences of non-library-level general access types when specifying a storage pool object if it supports subpools, because the "apparent" accessibility level is potentially deeper than that of the underlying object.

Tucker explains the rules and the reasoning. Essentially, we reject any cases that cause problems, at compile-time when we know that the pool supports subpools, and at run-time if we don't (Root_Storage_Pool'Class).

Tucker notes that "subpool access type" is a static notion, so it is wrong for the dynamic accessibility check.

So combine the rules proposed for 13.11.4(23) and 13.11(19):

When an access type with a specified storage pool is frozen (see 13.14), if the tag of the storage pool object identifies a storage pool that supports subpools, the following checks are made:

- the name used to specify the storage pool object does not denote part of a formal parameter nor part of a dereference of a value of a non-library-level general access type; and
- the accessibility level of the access type is not deeper than that of the storage pool object.

Program_Error is raised if either of these checks fail.

Gary says we need to write "non-library-level" in both rules (the dynamic rule is already correct).

Gary worries that there is a contract violation for a generic formal access type. We need the generic boilerplate after the Legality Rule.

Otherwise, after much investigation, we decide that there is no problem. "non-library-level" in the dynamic check might have to be determined dynamically in a (shared) generic unit.

Gary notes 13.11.4(23.a) says “exist” should be “exists” (“the type” is singular).

Approve AI with changes: 6-0-5.

AI05-0253-1/01 Accessibility of allocators for anonymous access of an object

Steve explains the problem with stand-alone objects of an anonymous access type (he coined SAOAAT for this use). The accessibility level of the last assigned object makes no sense for an allocator of a SAOAAT – there may not even be a last assigned object.

The accessibility level for such an allocator should be that of the declaration of the SAOAAT.

This modifies AI05-0148-1.

We'll assign the detailed wording to another subgroup.

Saturday afternoon, the subgroup proposes the following wording for the AI:

> !summary
>
> The accessibility level of an allocator assigned to a stand-alone object of an anonymous access type is that of the stand-alone object.
>
> !wording
>
> Insert immediately prior to the last sentence of 3.10.2 (14/3):
>
> For an anonymous allocator whose type is that of a stand-alone object of an anonymous access-to-object type, the accessibility level is that of the stand-alone object.

Steve notes that the accessibility of the type is not the same as the accessibility of the object. Tucker says it would be better to talk about the “declaration of the stand-alone object”.

“For an anonymous allocator whose type is that of a stand-alone object of an anonymous access-to-object type, the accessibility level is that of the declaration of the stand-alone object.”

Approve AI with changes: 10-0-2.

AI05-0254-1/01 Do we really have contracts right?

Tucker describes the changes to contracts made by AI05-0247-1. He agrees with Randy's assertion that the broad outline of the changes is what we need, but we may need to polish the wording.

In 6.1.1(10/3), delete “implicitly declared” as it is redundant. Tucker would like to restructure this. Randy notes that it is identically organized to 3.9.3(4-6). We'll let Tucker restructure this in a subgroup.

We don't need similar rules for postconditions, as they are always stronger on the new routine than on the original body. Thus we can just add the new ones at runtime without any bizarre effects.

There should be “unspecified” in the index (for 6.1.1(23-24)).

Various typos in 6.1.1(26/3) “of the of the”. “by the those”. “{callable} entity”.

Erhard is worried about where this implicit body (defined by 6.1.1(26/3)) is declared; that doesn't matter for these checks.

Tucker suggests that 6.1.1(26-27/3) just talk about the class-wide postcondition check, since it doesn't have any effect on anything else. Tucker would like to work on this wording; we'll do that in a subgroup.

Erhard would like to say “specific precondition and postcondition check, as well as the class-wide postcondition check” in the first sentence of 6.1.1(26), to make it clear that one case is missing.

We defer further discussion until after a subgroup has drafted replacement wording for these two sets of paragraphs.

On Saturday, we discuss the wording provided by the wording subgroup:

For 6.1.1(10-12),

> If a type T inherits homographs from a parent type T1 and a progenitor type T2, and
>
> - the corresponding primitive subprogram P1 of type T1 is neither null nor abstract; and
> - a class-wide precondition applies to the corresponding primitive subprogram P2 of T2 that does not fully conform to any class-wide precondition that applies to P1, then:
> - If the type is abstract the implicitly declared subprogram P is *abstract*.
> - Otherwise, the subprogram P *requires overriding* and shall be overridden with a nonabstract subprogram.

J-P suggests:

“If T is abstract...” instead of “If the type is abstract...”.

Tucker suggests that if the code you are inheriting has a class-wide precondition of True, then you don't have to override it (anything coming from the interface has to be stronger). In this case, counterfeiting is impossible, so if we use the interface's precondition, nothing bad will happen. This is likely to be common.

Add a bullet 1.5: * A class-wide precondition of True (implicitly or explicitly) does not apply to P1; and

Jeff notes that there is never a subprogram P defined, even though it is used here.

Tucker works for a while on this problem.

Replace 6.1.1(10-12) with:

> If a type T has an implicitly declared subprogram P inherited from a parent type T1 and a homograph of P from a progenitor type T2, and
>
> - the corresponding primitive subprogram P1 of type T1 is neither null nor abstract; and
> - the class-wide precondition True does not apply to P1 (implicitly or explicitly); and
> - a class-wide precondition applies to the corresponding primitive subprogram P2 of T2 that does not fully conform to any class-wide precondition that applies to P1,
>
> then:
>
> - If the type T is abstract the implicitly declared subprogram P is *abstract*.
> - Otherwise, the subprogram P *requires overriding* and shall be overridden with a nonabstract subprogram.

“{there is} a class-wide precondition {that} applies”...

Tucker now sends a replacement for 6.1.1(26-27):

> For any subprogram or entry call (including dispatching calls), the specific precondition and postcondition checks that are performed are determined by those of the subprogram or entry actually invoked. The class-wide postcondition check that is performed on a call of a primitive of type T includes class-wide

postconditions inherited from progenitors of T [Redundant: , even if the primitive is inherited from a type T1 and these postconditions do not apply to the corresponding primitive of T1].

We now take you to 3.9.2(20.4):

otherwise, the action is the same as the action for the corresponding operation of the parent type or progenitor type from which the operation was inherited {(except that additional class-wide postconditions may apply -- see 6.1.1)}. If there is more than one such corresponding operation, the action is that for the operation that is not a null procedure, if any; otherwise, the action is that of an arbitrary one of the operations.

Back to 6.1.1(27):

For a call via an access-to-subprogram value, the class-wide precondition check performed is determined by the subprogram or entry denoted by the prefix of the Access attribute reference that produced the value. In contrast, the class-wide precondition check for other calls to a subprogram or entry consists solely of checking the class-wide preconditions that apply to the denoted entity (not necessarily the one that is invoked).

Randy notes that “invariants” may also apply to null procedures. That will be handled by the invariant AI (AI05-0250-1).

We get into a discussion about conventions of non-Ada. Erhard suggests that we shouldn't allow them at all. Tucker says those are very important for static analysis, since the body can't be seen in that case. Randy worries that the implementation might not check them if they are assuming they are in the body. Bob says that falls under B.1(38.1/2). We should have an AARM note that for imported/exported routines, that paragraph applies (the checks may or may not be done) – compilers aren't required to do these checks at the call site just because the convention is not Ada.

Tucker has re-written 6.1.1(27) based on an unrecorded suggestion:

The class-wide precondition check for a call to a subprogram or entry consists solely of checking the class-wide preconditions that apply to the denoted callable entity (not necessarily the one that is invoked).

For a call via an access-to-subprogram value, all precondition and postcondition checks performed are determined by the subprogram or entry denoted by the prefix of the Access attribute reference that produced the value.

Approve AI with changes: 9-0-3.

(Ed says he would like to vote for paragraphs 10-12, and abstain on paragraphs 26-27. That's counted as an abstain.)

AI05-0255-1/01 Missing text about user-defined iterators

John: “A quantified expression {introduces}[is]...”

The subject should be broadened to cover quantified expressions as well as user-defined iterators.

Fix editor's notes: “is list is”

Erhard would prefer to put this after “loop_statement”, since it is related. He thinks the idea is that they go from the most important to the least important (most normal to unusual). So put it after 8.1(4).

Tucker agrees with Randy that we don't need to make any change to 8.6(6.a). So drop that.

There is a bad index entry in 3.2.4(2/3). Don't need any change in 3.2.4(16) (this rule only applies to discrete subtypes, none of the other cases contain a discrete subtype).

Bob notes “stamdard” in this AI.

Approve AI with changes: 13-0-0.

Erhard wants to add “The same applies to a loop parameter declared by an iterator_specification. “ to note 7 (5.5(11)). Randy thinks this would be more useful in 5.5.2, where this is declared.

Bob and Tucker would just as soon not worry about this; anyone writing user-defined iterators is already experienced with Ada.

So we won't make any change here.

AI05-0256-1/01 RCI units should not allow types with user-defined stream attributes

We discuss the problem. We don't want any stream attributes of visible types (directly or indirectly) that depend on remote subprograms (or subprograms declared in the private part that aren't remote but would be needed to do marshalling).

Is an instance special (which is what Tomas is asking)? Randy points out that an instance in the visible part is a remote subprogram by E.2.3(7/3) – that is clear. We don't want to treat instances in the private part differently. So we are not going to change this.

What should the rule be? Banning stream attributes in such a package is a bit strong, but easy. Only doing it on visible types isn't enough because a type in the private part can be used as a component of a visible type.

The details are left to a subgroup.

After splitting into subgroups, the following AI wording was presented:

> !summary
>
> RCI units do not allow types with user-defined stream attributes.
>
> !wording
>
> Add after E.2.3(16):
>
> Specification of a streaming attribute is illegal in the specification of a remote call interface library unit.
>
> Standard boilerplate:
> In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

There is nothing subtle about this. Erhard notes that “streaming” ought to say “stream-oriented”.

Tucker wonders what it means to instantiate something that is not RCI in an RCI. It's weird to change the categorization of a unit. On the other hand, if the package was written as a nested unit, the subprograms would be remote. Tucker suggests that renames of a package withed in packages is not going to change categorization, so both cases occur.

Randy objects that allowing anything in such an instantiation could allow the use of types that aren't remote as the parameter types of a remote subprogram. Someone suggests that it would be OK to allow remote types, but not others. It would make sense to say that you can only instantiate a package of “compatible” categorization in an RCI. But that's probably incompatible (there is no such rule now).

This is too complex to solve now, so we will requeue the AI on the entry of a task that has not yet started: Ada 2012 AIs.

AI05-0257-1/01 Insert returning a Position

The question is what happens when Insert that returns a Position is called with a Count = 0.

Bob Duff thinks that Matt's suggestion is reasonable. He suggests that you would insert the number of elements, and loop afterwards through the elements, then insert where you gave up. In that case, the semantics seems to work. Bob also notes that this case is unimportant.

Randy argues that this is error-prone. If you want to insert a number of elements and then immediately modify them, it is much safer to iterate inserting a single element at a time – because it is hard to write that following loop correctly. And it would only matter for performance for the vector, all of the other containers are pointer-based structures. In that case, the loop would zero-trip when the Count is zero. He thinks that the only safe case is when Count = 1 for this routine, any other use is error-prone or silly (if the Position is not used afterward).

We enumerate the possibilities of what Position could be when Count = 0:

(1) Position = Before (Matt's suggestion)

(2) Position = No_Element

(3) Raise Exception.

(4) Remove Count parameter (it is defaulted to 1). (Randy's preference.)

For Vectors, (1) was used (by accident or on purpose). So there is a inconsistency for (2) and (3). (4) is incompatible if the parameter is actually used; there is little evidence that it is used in practice.

Bob asks what happens to Before when you insert. For vectors, it becomes a bounded error, for the other containers, it stays valid.

We take a "can live with" straw poll: (1) 8; (2) 5; (3) 0; (4) 5.

That's not very conclusive, so we take a second vote asking which solution is preferred: (1) 7; (2) 0; (4) 3.

We'll let a subgroup work out the wording to this. On Saturday afternoon, the subgroup proposes the following wording:

> A.18.2(156) and (162) do not need any change. Add a Ramification after 162: “Thus, if Count equals 0, Position will designate the element designated by Before.”
>
> A.18.3(92) and (94) should say:
>
> “Position designates the first newly-inserted element{, or if Count equals 0, then Position is assigned the value of Before}.”
>
> A.18.10(143/3) and (145/3) also should use this wording.
>
> Remove A.18.10(143.a/3).

Randy notes that this AI should be a "binding interpretation" since it applies to an Ada 2005 container (lists).

Approve AI with changes: 12-0-0.

AI05-0258-1/01 Preserving information in an exception occurrence

After discussion, we decide that the problem is that the introductory text seems to describe all of the possible ways to raise an exception. It's not clear that it does, and there isn't any value to it doing so.

Erhard suggests replacing 11(2/3) by:

An exception can be raised explicitly (for example, by a raise_statement) or implicitly (for example, by the failure of a language-defined check).

Then this doesn't promise that it describes all of the ways to raise an exception.

Approve AI with changes: 12-0-0.

AI05-0259-1/01 Default convention of anonymous access-to-subprogram parameters

"-to-" is missing from the subject.

6.3.1(13.1/2) actually answers this question outright, so we have no idea what Adam was reading (or not reading). So write this as a confirmation.

Approve AI with changes: 12-0-0.

AI05-0260-1/01 Pedantic question on the wording of the definition of modulo

Approve AI: 11-1-0.

Bob Duff says he voted against because he does not believe in fixing the reference manual when not necessary.

AI05-0261-1/01 Default storage pools for storage pools

Brad would like to use the standard storage pool (heap) in a user-defined storage pool; and then use that storage pool as the default storage pool for all allocations. There is no way to refer to the standard storage pool for that purpose.

The usual trick of declaring an access type and using 'Storage_Pool of that doesn't work when the default storage pool is defined.

Bob notes that we are not talking about the default storage pool that an implementation would choose for any type, but rather a storage pool that could have been provided for some such type. That is, it shouldn't matter if a different pool is used by default in certain cases.

Jean-Pierre suggests that it would be the storage pool for access to Storage_Element. Bob worries that that would not get alignment right.

Tucker notes that GNAT has a pool package called System.Pool_Global which provides the sort of functionality we are talking about.

The type is called Unbounded_No_Reclaim_Pool and there is an object called System.Pool_Global.Global_Pool_Object.

It seems reasonable to adopt something like this.

We'll do that after Ada 2012 is finished. This is something that implementors can add on their own (the necessary permissions already exist in the Standard), and surely if there is a "standard" definition, many implementors will adopt it. It can then be added to the next version of the Standard.

AI05-0263-1/00 No_Dependence should sometimes be excluded from applying to the runtime

Brad had asked whether "No_Unchecked_Deallocation", "No_Unchecked_Conversion", which are obsolescent, should be mentioned in H.4(24).

Bob thinks that this sentence is rather silly, if the run-time system is not written in Ada.

No_Dependence is not in the index, that needs to be added.

Add: "or the equivalent use of No_Dependence" to this rule. Otherwise, "No_Dependence (Unchecked_Deallocation)" could not be ignored in the run-time system, which seems to break the intent.

Approve AI with changes: 10-1-0 (Bob voted against because he does not want to change this junk rule)

[Editor's note: This AI was "virtual" during the meeting; it was created afterward to capture this discussion.]