Minutes of the 48th ARG Meeting

6-9 December 2012

Boston, Massachusetts, USA

Attendees: Steve Baird, Randy Brukardt, John Barnes (but not early Saturday morning), Erhard Ploedereder, Ed Schonberg, Tucker Taft, Bob Duff, Gary Dismukes, Brad Moore, Geert Bosch (but not Sunday or late Saturday), Jean-Pierre Rosen (via Skype, Thursday late, Saturday, Sunday).

Observers: Dan Eilers (Thursday only).

Meeting Summary

The meeting convened on Thursday, 6 December 2012 at 14:00 hours and adjourned at 12:12 hours on Sunday, 9 December 2012. The meeting was held in the Nantucket room at the Hyatt hotel in Boston. The meeting covered most of the agenda.

AI Summary

The following AIs were approved:

AI05-0299-1/01 Last-second presentation issues in the Standard (9-0-1)
AI12-0039-1/01 Syntax for membership expressions is ambiguous (9-0-1)

The following AIs were approved with editorial changes:

AI12-0022-1/03 Raise expressions for specifying the exception raised for an assertion (9-0-1)
AI12-0027-1/04 Access values should never designate unaliased components (8-0-2)
AI12-0028-1/02 Variadic C functions (9-0-2)
AI12-0037-1/01 New types in Ada.Locales can't be converted to/from strings (7-0-3)
AI12-0038-1/01 Shared_Passive package restrictions (5-0-4)
AI12-0040-1/01 Resolving the selecting_expression of a case_expression (9-0-1)
AI12-0043-1/01 Details of the storage pool used when Storage_Size is specified (10-0-1)
AI12-0045-1/01 Pre- and Postcondition are allowed on generic subprograms (11-0-0)
AI12-0046-1/01 Enforcing legality for anonymous access components in record aggregates (8-0-2)
AI12-0047-1/02 Generalized iterators and finalization of the associated object (11-0-0)
AI12-0048-1/01 Default behavior of tasks on a multiprocessor with a specified dispatching policy (8-0-2)
AI12-0049-1/01 Invariants need to be checked on the initialization of deferred constants (9-0-1)
AI12-0051-1/01 The Priority aspect can be specified with Attach_Handler (10-0-0)

The intention of the following AIs were approved but they require a rewrite:

AI12-0001-1/02 Independence and Representation Clauses for atomic objects (9-0-1)
AI12-0003-1/02 Default storage pool for storage pools (9-1-0)
AI12-0016-1/01 Implementation model of dynamic accessibility checking (9-0-1)
AI12-0032-1/03 Questions on 'Old (8-0-2)
AI12-0035-1/02 Accessibility checks for indefinite elements of containers (11-0-0)
AI12-0050-1/01 Conformance of quantified expressions (11-0-0)
AI12-0052-1/01 Implicit objects are considered overlapping (7-0-2)

The following AIs were discussed and assigned to an editor:

AI12-0020-1/01 'Image for all types
AI12-0033-1/02 Sets of CPUs when defining dispatching domains
AI12-0042-1/01 Type invariants cannot be inherited by non-private extensions
AI12-0044-1/01 Calling visible functions from type invariants expressions
AI12-0054-1/02 A raise_expression does not cause membership failure

The following AI was discussed and voted No Action:

AI12-0053-1/01 Predicate failure raises Constraint_Error (10-0-1)

The following SI was approved with editorial changes:

SI99-0065-1/02 Generic_Actual_Part and null procedure defaults (6-0-4)

The following SIs were discussed and assigned to an editor:

SI99-0062-2/01 Forget the Semantic Subsystem
SI99-0066-1/01 Variability between implementations revisited

Detailed Minutes

Previous Meeting Minutes

As usual, John has a few editorial comments. No one else has any changes.

Approve minutes: 11-0-0.

Date and Venue of the Next Meeting

Next meeting: Berlin, Germany, June 14-16, in association with the Ada-Europe 2013 conference. Erhard will distribute information about the meeting logistics. WG 9 will get a piece of the available time, we'll meet the rest, finishing up in early afternoon on the 16th.

Next year's HILT conference will be in Pittsburgh, Pennsylvania. No dates have been set yet, but we'll plan to meet afterward as usual.

Editor's Draft of the Standard

Randy notes that he will soon be starting to add approved AIs to the working draft of the Standard. He needs this at a minimum for ACATS work (he needs to be able to easily see if a rule has been changed).

He notes that this is the time to make changes to the layout of the Standard, if we want to make any.

Should we make this draft public via ada-auth.org? Or available only to the ARG? It seems that there is no good reason to keep it hidden, so long as it is clear that it is a working document and not the current Ada Standard. Mark the paragraph numbers in the same style (that will be /4).

Randy wonders if he should make !corrigendum sections. We don't need them unless we decide to do a Corrigendum or Amendment document. But having them makes the formatted AIs easier to read (because the formatting is always accurate with them), and we have tools to compare this text with the draft Standard. This tool does catch quite a few errors. The ARG has no opinion on whether to do this or not.

Randy wonders if the draft AARM should show fewer changes; some paragraphs are unreadable because they've been changed repeatedly. Yes, only show changes from Ada 2012, as the standard was **revised** this time. (Also keep a separate version that shows them all, probably make this one downloadable.)

On Friday, Randy asks whether the Springer version should fixed for the **in** parameter invariant check (AI12-0044-1). He notes that the changes are relatively small, and most people will use that edition and the on-line one, rather than the "official" ISO standard. Changing those editions would greatly reduce the concern about users being confused. Some suggests adding a footnote or some note like that, rather than changing the normative text. Bob objects to even that, we know there are bugs and surely we can't fix all of them. Besides, we're going to look at a permission to check **in** parameters so long as they are not evaluated as part of the invariant. In that case, the difference between the existing text and the reality could be fairly small.

We eventually decide to leave the printed standard (and the web based versions) unchanged for the AI12-0044-1 changes.

Amendments to Ada 2012

Randy had included the following position statement in the agenda:

> Until we (the ARG) are officially charged (by WG 9) with creating a revision or other update to the Ada Standard, we should not be approving or rejecting Amendments. A number of people have complained about what appears to be haphazard inclusion of pet ideas into the Standard; it would be best to look at the ideas as more-or-less a whole rather than piece-meal good ideas. The ARG can work out Amendment ideas (especially valuable if some implementers want to implement them early), but should hold off on final approval. Ideas we don't like can be put on hold until we start working in earnest.

Bob strongly agrees with this position, he notes that he had sent an e-mail to that effect yesterday.

Bob suggests having a status to specify that we have worked on items and that we think they're fully baked. “Promising” is suggested, means hold and probably in, “Hold” is probably out (and probably half-baked at most).

Thanks

We thank SIGAda for the accommodations.

Old Action Items

Steve Baird did not create a proposal for AI12-0020-1 (but he requests to talk about it to determine the interest in his current ideas). He also did not propose a checked specification that a subprogram is non-blocking. Ed Schonberg didn't work on AI12-0002-1. All other action items were completed.

New Action Items

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

Almost Everyone:

- Create solutions for issues created in ASIS by Ada 2012. (Use the implementation in ASIS-for-GNAT as a starting point, if possible.) Assignments to be circulated ASAP.

Steve Baird:

- AI12-0003-1 (determine wording for "immediate scope" of a pragma)
- AI12-0016-1
- AI12-0020-1
- AI12-0032-1
- Proposing checked specification that a subprogram is non-blocking (see AI12-0026-1 for June meeting).

Randy Brukardt:

- AI12-0052-1
- AI12-0055-1 (create empty AI with question, forward to IRTAW, see AI12-0048-1)

Editorial changes only:

- AI12-0022-1
- AI12-0027-1
- AI12-0028-1
- AI12-0037-1
- AI12-0038-1
- AI12-0040-1
- AI12-0043-1

- AI12-0045-1
- AI12-0046-1
- AI12-0047-1
- AI12-0048-1
- AI12-0049-1
- AI12-0051-1
- SI99-0065-1

Gary Dismukes:

- AI12-0035-1

Bob Duff:

- AI12-0001-1
- AI12-0054-1

Erhard Ploedereder:

- AI12-0003-1 (other than task assigned to Steve Baird)

Jean-Pierre Rosen:

- Work with Sergey Rybin to find and document "useful" extensions to the capabilities of the standard that exist in ASIS-for-GNAT.

Ed Schonberg:

- AI12-0002-1
- Extract Ada 2012 queries from the ASIS-for-GNAT source and distribute them to the ARG as a starting point for their ASIS homework.

Tucker Taft:

- AI12-0042-1
- AI12-0044-1
- SI99-0062-2 (unwinding the introduction; this part will have a new SI number)
- SI99-0066-1

IRTAW (Alan Burns and Tullio Vardanega will submit)

- AI12-0033-1
- AI12-0055-1

Detailed Review

The minutes for the detailed review of AIs and SIs are divided into ASIS Issues (SIs), Ada 2005 AIs (AI05s), and Ada 2012 AIs (AI12s). 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.

If a paragraph number is identified as coming from the Ada 2012 Standard, the number refers to the text in the final version of the Ada 2012 AARM. Paragraph numbers in earlier drafts may vary.

Detailed Review of Ada 2005 AIs

AI05-0299-1/01 Last-second presentation issues in the Standard

Randy notes that these changes were made to the final draft that ISO will publish next week, so any substantive changes will have to handled in a future presentation AI. Erhard notes an extra "and" in the scope statement; he sends an e-mail to Randy for future processing. Nothing else is noted.

Approve AI: 9-0-1.

Detailed Review of Ada 2012 AIs

AI12-0001-1/02 Independence and Representation Clauses for atomic objects

Bob and Tuck argue that the Recommended Level of Support is wrong as it does not match the AARM Ramifications. [Editor's note: I didn't record what ramification(s) they referred to. I can't find any that clearly conflict with the Recommended Level of Support; the only one that might be read that way is 13.2(9.a), which says that an aliased component won't get packed very tightly because "its Size will generally be a multiple of Storage_Unit". But this statement appears to be circular, as the Size of a component is determined by the amount of packing applied, so essentially says that an aliased component won't get packed tightly because it won't get packed tightly. It would make some logical sense if was meant to refer to the Size of the subtype of the component, but then it is just wrong because the Size of a subtype is not affected by aliasedness.]

Geert notes that Pack indicates that the components are not independent; that makes no sense with Atomic. We scurry to the Standard to see what it actually says.

9.10(1/3) discusses independence, and it says that specified independence wins over representation aspects, so there is no problem there.

C.6(8.1/3) should include aliased in things that cause “specified as independent”.

Tucker thinks C.6(13.2/3) is misleading, as it seems to imply packing is not allowed when independence is specified.

The Recommended Level of Support for Pack needs to be weakened to allow atomic, aliased, and so on to make the packing less than otherwise required.

Bob will take this AI.

Approve intent: 9-0-1.

AI12-0003-1/02 Default storage pool for storage pools

“STandard” should be “Standard”.

The paragraph reference 13.11.3(3.1/3) in "change 13.11.3(3.1/3)" in !wording should be 13.11.3(3.2/3).

Bob still wants to give the default storage pool a name. Randy notes that you could then pass it as a parameter into a scope. Erhard notes that this is not the same as giving a pool; if there is special optimized mechanism, you would not get that.

Bob says that a compiler could use a very general pool for this name, and when the name is directly used, you get the optimized version. If passed as a parameter or the like, they have to get the general mechanism. Randy is dubious that compilers would go to that much trouble – they're much more likely to use the general pool everywhere.

If there was such a name, where would it be defined? That's not obvious. So that's still pretty kludgy. If we don't need such a name, we avoid that problem. The pragma is based on the existing Default_Storage_Pool mechanism, there is nothing new here. All this does is allows the programmer to revert the Default_Storage_Pool to the initial state when necessary.

Erhard notes that “immediate scope” is not defined for pragmas. That should be fixed. Or do we need it at all? Other pragmas don't use the term “immediate scope”; Suppress uses "region".

Erhard asks whether there should be a consistent way to handle conflicts for configuration pragmas. We don't think that a general mechanism would work, the pragmas are all different. Randy notes that pragma Suppress and Unsuppress are configuration pragmas with well-defined and complex conflict semantics – any general solution would have to avoid breaking those semantics. Doing so might eliminate much of the advantage to having a general rule. There is an argument that someone should check if conflicts are well-defined for existing configuration pragmas, but generally we aren't trying to make work for ourselves.

Steve will determine how much wording would be needed to define “immediate scope” for pragmas.

Erhard will check whether the region concept could be be used for these pragmas.

Approve intent of AI: 9-1-0. Bob does not agree with the approach in the AI (he would rather use a named general pool).

AI12-0016-1/01 Implementation model of dynamic accessibility checking

We look at the example Accessibility_Test.

The stack frames would look like (the stack grows down here):

```
----
Accessibility_Level [Static Level 1, Dynamic Level 1]
----
P1 [Static Level 2, Dynamic Level 2]
----
Call_Proc (P2'access, null, null); [Static Level 2, Dynamic Level 3]
Local1
----
P2 (Local1'access, null); [Static Level 3, Dynamic Level 4]
type Ref is access all Integer with Storage_Size => 0
Ptr : Ref;
----
Call_Proc (P3'access, Local1'access, null); [Static Level 2, Dynamic Level 5]
Local2
----
P3 (Local1'access, Local2'access); [Static Level 4, Dynamic Level 6]
Ptr := Ref (X == Local1'access); – Should be OK
Ptr := Ref (Y == Local2'access); – Should fail
```

In this case, the static levels for Local1 and Local2 are the same, so it is impossible to get different results for this check using only static levels.

Thus the "small integer" model is doomed to failure.

The dynamic levels are different, and give accurate answers. But a naive implementation has distributed overhead (an extra parameter on every call).

The solution is to represent an accessibility level as a reference to a “Level_Obj”. These contain enough information to make the check. The exact information contained depends on the solution chosen.

Steve and Tucker go on to explain the solution given in the e-mail.

In this solution, a Level_Obj represents a set of accessibility objects that have a longer lifetime than you.

Geert notes that a simple stack check works if you are in a single task and you have a contiguous stack. This matches the dynamic levels.

Randy notes he sent a fuller proposal on this idea in e-mail. Its worst case performance would be bad, but it would be nearly pathological to happen (a recursive routine creating tasks with local tagged types which are being returned?).

3.10.2(22.x) needs to be replaced by something. The only thing that matters is that there exist decent algorithms, the AARM doesn't need to explain them. Probably the AARM should simply say that algorithms exist, and point at the AI for longer explanation.

It would be better if these models were explained in a proper paper, given at a conference. It's too late to do that for Ada-Europe (the deadline is only a few days away), but HILT '13 would be appropriate.

Steve will revise the wording of the AARM note. The note will assume that some paper exists to explain these implementation models. He also will outline the paper (to be authored with Randy).

Randy notes that this will be a Ramification, and the AI should be updated accordingly. (There is no normative wording changes that will be proposed.) He also notes that Steve needs to write up a longer explanation of the problem. The AI ought to be understandable without spelunking in the e-mail, and in any case the e-mail is a lot longer than necessary: all we need are a couple of examples showing that the "small integer model" doesn't work.

Approve intent of AI: 9-0-1.

AI12-0020-1/01 'Image for all types

Steve would like to discuss this one a bit. He didn't write up anything because he thinks a different direction is needed. He says that early discussions on this led to noting that this needs some ability for customization. That quickly turns into a mess, at least with the stream-based solution originally discussed.

Later, he started investigating an interface for traversing structures in your programs. (For instance, components of an object of a composite type). An interface with many callbacks was tried but it pretty messy. He's now looking more at something more like an active iterator.

He wonders if this is worth pursuing. The group says that it sounds interesting and he ought to continue.

Steve will try to work something out.

AI12-0022-1/03 Raise expressions for specifying the exception raised for an assertion

Should this AI being a Binding Interpretation? Via the rules we just adopted (see "Amendments to Ada 2012"), we cannot adopt this as an Amendment for several years, and the need for some solution to this problem is immediate.

Erhard wonders about the precedent of adding syntax as a Binding Interpretation. We don't want this to a set a precedent; this was discussed in Kemah and we decided then we wanted a solution, we just hadn't decided exactly what it was.

Bob would like to allow no-return procedures in this context. He says that their purpose is similar to that of a raise expression. There is concern that this usage would be ambiguous with similar function calls and thus might be incompatible. In any case, that is way beyond what we previously discussed; as such, it should be proposed as a future Amendment.

Approve AI as a Binding Interpretation: 9-0-1.

AI12-0027-1/03 Access values should never designated unaliased components

Subject: “designate[d]”

Using Legality Rules to directly handle this problem is probably going to cause an incompatibility.

Moreover, there was no rule at all in the Standard to specify the accessibility of a value conversion. It certainly seems like it was intended to allow a value conversion to make a copy of the object (it seems to be required if the representation changes). In that case, the accessibility of the copy is going to be different than that of the original object. The idea is to use rules similar to those for aggregates.

The example in the question should be illegal, as the copy should be very short lifetime (and thus the accessibility check will fail).

“may” should be “might” in the dynamic wording for 4.6.

Add “redundant” in the square brackets.

"If there is a composite type that is an ancestor of both the target type and {the operand type} and then either..."

Tucker does not think that there is any need to prohibit making copies. Randy argues that there is a possibility of an incompatibility for by-reference types (he would except the by-reference cases from the short-lived accessibility). Tucker argues that this is too rare for that solution, let's keep this simple.

Bob points out that a value conversion of a limited by-reference type should never allow copying. So the by-reference part actually is needed in some form.

Tucker notes that a by-reference type passed as an in parameter still can be assumed to point at the real object (and see changes).

“and if the operand type {does not have aliased components}[is not such an array type].”

Bob does not like “required” and “prohibited”. It should just say that it “is copied” or “is not copied”.

Approve intent: 9-0-1

Steve will try to provide a rewording tomorrow.

AI12-0027-1/04 Access values should never designate unaliased components

Steve sent new wording while most of us were asleep Saturday morning. Tucker then sent a note suggesting removing “shall” from the Dynamic Semantics wording.

We clean up the wording:

Append at the end of Dynamic Semantics section of 4.6:

Evaluation of a value conversion of a composite type either creates a new anonymous object [(similar to the object created by the evaluation of an aggregate or a function call)] or yields a new view of the operand object without creating a new object:

- If the target type is a by-reference type and there is a type that is an ancestor of both the target type and the operand type then no new object is created.
- If the target type is an array type having aliased components and the operand type is an array type having unaliased components, then a new object is created.
- Otherwise, it is unspecified whether a new object is created.

Tucker would prefer appending this at the end of 3.10.2(10/3):

“Corresponding rules apply to a value conversion (see 4.6).”

Randy wonders if we need the 6.2(10/3) change at all, as this is a by-reference type, can a copy be made at all? Yes, unrelated array types can use a copy in that case.

Approve AI with changes: 8-0-2.

AI12-0028-1/02 Variadic C functions

Fix the question: “Should this note [should] be rewritten or deleted? (Yes.)”

No one understands where the value 16 comes from. Steve says that it exists just to give a minimum number of these identifiers supported. That doesn't seem worth the complication, especially as *any* support for *any* convention identifiers is implementation defined. Change the wording for B.3(60.15/3) to just say:

"The identifiers C_Variadic_0, C_Variadic_1, C_Variadic_2, and so on are convention identifiers."

Geert wonders what it means to Export an Ada function with one of these conventions. Erhard suggests that you just give Ada the actual number of parameters it wants, and the rest get dropped on the floor. That seems to work, and it makes sense (sort of). In any case, we don't want to prevent implementations from doing something useful, and we don't require support for such exports (or anything else, for that matter).

This convention should only be allowed on subprograms and access-to-subprogram (which is what the wording says). But the representation for access-to-subprogram types shouldn't be required to be the same as for convention C (if C represents them differently, we surely want the Ada compiler to be able to reflect that).

So the last sentence about type representation should be deleted from the wording for B.3(60.15/3)

Approve AI with changes: 9-0-2.

AI12-0032-1/03 Questions on 'Old

Tucker and Erhard would like a very simple wording that the type is the same as the original.

Steve and Bob point out that doesn't work. X'Old is a constant that has a different lifetime than X; it can't have the same accessibility. Access discriminants have weird accessibility; X'Old cannot have that accessibility.

Steve mentions a similar case: the accessibility of X'Old where X is a stand-alone object of an anonymous access type. The accessibility is not that of X (which is dynamic), but a copy of X at the point of entry.

Erhard wants X'Old to not be a real object so that we don't need to answer questions such as finalization and accessibility. But X'Old has to be able to be queried, passed as a parameter, and so on. It's hard to imagine how making X'Old some sort of shadow object could be made to work. He says if X'Old is a real object, it needs all complex rules to explain where it is declared, and so on. Sure, but if X'Old is not a real object, then we need rules to explain what it means when the underlying real object is changed, deallocated, and so on. It doesn't seem like there is any simplification to be found that way.

Typo in the !question, there is a missing arrow after Post “with Post {=>|}”.

Looking at the wording:

“qualified conversions” should be “qualified expressions”

“Each {occurrence of} X'Old” in the beginning of the wording.

Why are we not allowing a universal object? We have rules for runtime universal types. So we should just apply those. Kill that bullet.

Tucker suggests “(including anonymous and universal types)” in the final bullet. Bob would rather put that in the AARM. Tucker thinks that it is important to point out that this is not something you can do explicitly in an Ada program.

The postcondition should be inside of the protected action so that the object's state can be queried safely (as postconditions can be about describing properties created by this body).

Steve will update this AI again.

Approve intent of AI: 8-0-2.

AI12-0033-1/02 Sets of CPUs when defining dispatching domains

The second sentence of the wording for D.16.1(23/3) should say “Create” (upper-case C).

The section should be added to the paragraph numbers.

Steve wonders what happens you pass in a null range or an empty set. For instance, (1..3 => False), what are the bounds (values) returned by Get_First_CPU and Get_Last_CPU?

What happens if the domain is empty? Randy says its silly to create an empty domain, what happens to a task assigned to it? It can never run (as you can only set the Dispatching_Domain of a task once).

Tucker is concerned that you might have a configuration that contains no CPUs in some domain. He thinks it would be hard to work around that. Randy is dubious; you'd have to have conditional code to avoid assigning any tasks to the empty domain; it seems that it would not be much harder to have the same conditional code around the creation of the domain in the first place.

He suggests that we simply define the bounds for an empty dispatching domains. Probably they should work like strings, that is, 1 and 0 for Get_First_CPU and Get_Last_CPU.

Someone points out that 0 isn't in the range CPU, so Get_Last_CPU would have to raise an exception for an empty dispatching domain. That obviously won't do. Tucker suggests changing the subtype of Get_Last_CPU to CPU_Range. The changes seem to be spreading.

Randy suggests asking IRTAW whether they even want empty dispatching domains, or whether we should be requiring an exception to be raised when one is created. That meets with general approval.

Keep alive: 10-0-1.

AI12-0035-1/02 Accessibility checks for indefinite elements of containers

The laundry list of subprograms here is not liked by anyone.

Tucker suggests that the term should be “perform indefinite allocation”.

He also suggests adding a bullet similar to the following to each indefinite container in the static semantics:

- Insert, Replace_Element, and “&” perform indefinite allocation.

Where, of course, the routines in question are names.

The definition would remain in A.18. “Some operations perform indefinite allocation...” (or wording similar to that used for tampering).

There is a question about streams. Is it possible to stream in something whose accessibility is too shallow for the container? The check is against the element type of the container instance, and the contents have to live at least as long as that element type. Thus, anything we can stream out has already been checked. And thus anything that we can stream in is already OK. (Recall that we only promise to be able to stream in stuff that we stream out, not anything in general.) And we don't promise any extra interoperability for streaming of indefinite containers. (We do

for the bounded and unbounded forms, but that doesn't apply to the indefinite forms.) If there was an interoperability promise, a less nested instance could cause trouble.

Gary will update the AI.

Approve intent: 11-0-0.

AI12-0037-1/01 New types in Ada.Locales can't be converted to/from strings

The country code should be 'A'..'Z' (that is, upper case).

Add a paren ')' at the end.

Approve AI with changes: 7-1-2.

John says that he voted against as he feels this is an abuse of machinery.

Randy should add an AARM note to the RM to explain why these types are declared this way.

Steve says that this idiom is an issue with membership tests. Ed suggests that we just let it raise Assertion_Error. Tucker agrees after some thought.

```
type Language_Code is new String(1 .. 3)
   with Dynamic_Predicate => (for all E of Language_Code =>
      E in 'a' .. 'z');
type Country_Code is new String(1 .. 2)
   with Dynamic_Predicate => (for all E of Country_Code =>
      E in 'A' .. 'Z');
```

We turn to the semantic problem that Steve noticed: a subtype membership test that contains a “raise Constraint_Error” will raise an exception rather than return False.

We wonder if we should raise Constraint_Error rather than Assertion_Error for predicates. It's too late to do that. There is some dissent on that. It should be possible to change a constraint to a predicate (or vice versa) without changing the semantics. Perhaps we should reconsider?

Tucker suggests something weirder; for a predicate with a raise_expression in it, the raise_expression exists to specify the behavior on failure. He suggests that it should simply cause the membership to return False. This is rather weird.

Geert worries that using a raise inside of a function called from a predicate then would raise an exception when used in a membership.

These are separate AIs, Bob will draft them for consideration later during this meeting.

Approve AI with changes (without “raise Constraint”): 7-0-3.

After discussing AI12-0053-1 on Saturday, we briefly revisit this AI.

Should we change it to explicitly raise Constraint_Error? Probably not, no one should care what the exception is. Geert notes that this formulation allows using these subtypes in a membership expression, even if AI12-0054-1 fails.

Should we revote on this AI? No, we already voted three times [actually, only twice – Editor] and the conclusion seems unchanged.

AI12-0038-1/01 Shared_Passive package restrictions

“access type[s] declarations” in the !question

Tucker tries to explain this change. Essentially, we don't have enough restrictions on shared passive packages to account for the extensions made to declared-pure packages by Ada 2005. Specifically, we have rules to try to prevent access values that designate objects that belong to other partitions, as those may not exist as long as the shared-passive partition. Those rules don't work if the access type is not declared in a shared-passive package; since access types can be declared in declared-pure packages, we have a problem.

Turning to the wording: Randy wonders about the hyphen in “declared-pure package”. E.2.2 uses “declared pure package” in Ada 95 wording, while E.4 uses the hyphenated form. That's inconclusive. [The AARM notes in 10.2.1 use both "declared-pure package" and "declared pure library unit", which doesn't help – Editor.] It is suggested that the "proper" English form (with the hyphen) be used.

Randy asks if “in a declared-pure package” is good enough; he convinces himself that only named entities are possible, so that's all that needs to be covered. “within a declared-pure package” might be better.

Steve asks if the wording covers subtypes of types declared in a declared-pure package. No. Change this to “denotes an access {sub}type”.

Erhard objects to “includes a name”. He notes that this would make attribute prefixes illegal, as in `D : Integer := N'Size;.`

Tucker notes that these were not allowed in Ada 95, so there isn't much value to being very specific. So he was trying to cover everything.

Erhard wonders if library-level is enough. Tucker thinks it is, as nothing you can do inside a subprogram can cause a problem (because it is only the library-level changes that matter).

The section number E.2.3 should be E.2.1, everywhere.

Approve AI with changes: 9-0-2.

On Sunday morning, Tucker wants to revisit this AI. He says that the wording needs to have “part” inserted in various places. A (non-limited) record type could contain an access component.

Someone asks about private types. Non-limited private types need external streaming, and they don't have that if they have an access component. But there is no such requirement for limited private types. So there is a concern that declared pure packages could have a limited private type that contains an access value; if a constructor function passes an access value, that value could be stored. Ed notes that these are preelaborable, so that initializing with a constructor function is illegal. So this isn't a problem.

Re-word this as:

> it shall not contain a library-level declaration of an access type that designates a class-wide type, task type, or protected type with entry_declarations{; further, it shall not contain a library-level declaration that includes a name that denotes a subtype with a part having an access type that is declared within a declared-pure package}.

A non-limited private type that has stream attributes and an access component could be legal in a declared-pure package (as the type would then have external streaming). If the package also declares a function that is passed an access value and returns that type could cause trouble by storing the parameter value into the component and then returning that to be stored in an object of the shared-passive partition. No solution is obvious: a solution would seem to require making referencing any non-limited private types illegal which seems way over the top. We're going to ignore this problem. [Did we really decide this? It seems pretty dubious: either we have to have a Legality Rule about it, it has to be detected at runtime, or it has to be erroneous to designate a different partition. *Something* has to happen in this case, otherwise it is expected to work (and in that case, we don't need these other rules, why only allow it via a silly end-run?). - Editor.]

Approve AI with changes: 5-0-4.

AI12-0039-1/01 Syntax for membership expressions is ambiguous

We discuss whether these prefixes are best, Tucker thinks that there is some pattern to the meaning of prefixes in the Standard. A few minutes of looking through the syntax summary of the Standard makes it clear that there isn't any significant pattern. So the use of the prefixes is fine.

Approve AI: 8-0-2.

AI12-0040-1/01 Resolving the selecting_expression of a case_expression

This has to be a complete context so that the case labels don't participate in resolution. The resolution wording for assignment_statement and case_statement are almost identical, but for the former we want everything to participate and for the latter we don't, so clearly there has to be a difference, and that is that a *selecting*_expression is a complete context.

There is a missing “to” in the first paragraph of the !discussion.

Approve AI with change: 9-0-1.

AI12-0042-1/01 Type invariants cannot be inherited by non-private extensions

Randy suggested to avoid introducing holes into invariants by making it illegal to visibly derive from a private type that has a class-wide invariant. Otherwise, it is relatively easy to have the inherited invariant to depend on visible state, at which point all of the guarantees are lost.

Tucker changes the informal position he had previously and announces that this doesn't work. He points out that this still allows deriving a private type from a visible record type and adding an invariant to the private type. An invariant should never depend on visible state, but there is no sane way to check that. Randy notes that that could also be illegal; we have a similar (but reversed) rule for extension aggregates.

Jean-Pierre comments that a dispatching call could cause problems. His example is similar to the one Randy sent in his e-mail of Nov 30.

Randy comments that the main reason he suggested this rule is that the rules for invariant checks assume that the type is a private type. He thought that a Legality Rule would make a relatively easy fix; if it proved to be overly restrictive, we could invent new rules and remove the Legality Rule. Such a change would be compatible; but if we get the checking rules wrong, we're hosed forever.

Bob asks what the hole(s) in the checking rules are. We don't know that checks are done properly for extensions that are visible rather than private, and especially for extensions that aren't in packages.

A quick look at the wording does not turn up any problems for visible extensions in package specifications; the rules seem to work as written.

We look in detail at extensions in subprograms. The inherited routines are checked, because the original routines are called in that case (that is the point of the note 7.3.2(24/3). Overridden routines aren't checked by the rules given here. But that should be OK, because an overridden routine cannot violate the original invariant – it has to call something that will check the invariant. Randy is skeptical of this assertion because of dispatching, especially inside an invariant expression. Others would like him to come up with an example off-line, and the discussion moves on.

[Editor's note: An important part of this problem is that it's not clear to me when a class-wide invariant (or precondition or postcondition) is dispatching. One would hope that statically bound calls would also have statically bound invariants/preconditions/postconditions, but there is nothing in the Standard which could be construed that way. It appears to me that these expressions are effectively re-dispatching, and that causes all kinds of problems. In particular, it means that a set of specific invariants (or other contracts) is not equivalent to the similar class-wide invariant. Perhaps this should be handled as a separate problem, but it is clearly related here.]

Randy also was concerned about the effect of routines inherited with different visibility than they originally had. That could cause a private routine to gain invariant checks in an inherited subprogram. That could be bad, especially if there is a dispatching call from a body that is expecting invariant checks to not be made on this routine.

Most of the group is confused. We look at the following example of the problem:

```
package P is
   type T is ...
      with Type_Invariant'Class => E1;

private
   procedure PP (X : in out T); -- Does not check E1
end P;

private package P.C is
   type T2 is new T ...
      with Type_Invariant'Class => E2;
   -- Inherits procedure PP (X : in out T2) here, the rules imply that E2 is checked on this routine.
end P.C;
```

This is low priority if the only problem is in private children. It's unclear if that is the case (it is the obvious example, and no one has tried very hard to come up with examples), and in any case we should get this right if reasonably possible.

Steve has a wording problem in this area. It is in the mail, August 3, starting with “strictly” speaking. Essentially, the wording only works for the first level of inheritance. Further levels don't work (unless one invokes the Dewar rule). That should be addressed as well.

Keep this alive, and give it to Tucker.

AI12-0043-1/01 Details of the storage pool used when Storage_Size is specified

Correct the summary to remove the redundant word in “must must”.

Tucker thinks that the Storage_Error should be raised at the freezing point of the type, not at the attribute_definition_clause. That seems to be an Ada 95 bug. [Also note that if Storage_Size is specified using an aspect, there is no attribute_definition_clause to raise an exception at, so it is even more wrong for Ada 2012 – Editor.]

...If the implementation cannot satisfy the request, Storage_Error is raised at the {freezing} point of {type *T*}[the attribute_definition_clause].

Comma police: drop the comma in “...*T*, or other...” in the penultimate sentence.

Jean-Pierre asks that the last sentence of the discussion section be corrected: “someone {who}[that]...”

Remove the parenthetical remark at the end of the !discussion.

Approve AI with changes: 10-0-1.

AI12-0044-1/01 Calling visible functions from type invariant expressions

Erhard thinks that defining “inner” and “outer” calls would solve this problem. There is much doubt that such a scheme would work, mainly because of the complex visibility rules of the language.

Erhard is asked to explain his idea further. He shows an example.

```
type T is tagged ...
   with Type_Invariant => ...F1...; -- "inner" call
```

```
function F1 (Obj : T...);
```

The call to F1 in the invariant expression is an “inner” call, there would be no check.

```
function F1 (Obj : T) is
begin
   ... F2 (...); -- in the body, “inner” call.
end F1;
```

This is big change in the model. A compiler probably would have to generate to two bodies for many routines (invariants are checked in body).

Steve says he prefers B or C from the AI.

There doesn't seem to be much agreement; everyone seems to have a favorite solution, and they're all different.

We make a list of the most popular solutions.

(C) as in original AI.

(G) Erhard's idea of "inner" calls not checked (see above).

(H) Check on procedure **in**, no check on **in** for function, use postcondition or **in out** if a check is required. (Can get infinite recursion from body when calling procedures in function called from an invariant).

(I) No checks on **in** anywhere. The objection here is that many **in** parameters get (logically) modified, if they're in packages structured like Text_IO.

(J) Turn off **in** for all, have aspect to turn on.

(K) Erhard suggests categorizing a function only as a reader. (It does not modify any state.) Such a rule is generally useful. Steve asks if this is checked by the compiler.

(K[I]) – Checked;

(K[II]) – Unchecked.

[Note: Many of these options will include a Legality Rule to detect cases where the direct invariant expression calls a routine that would cause a recursive check.]

Straw poll: which of these is your favorite?

C – 0; G – 3; H – 4; I – 1; J – 0; K – 2.

Straw poll: which of these are your top 3? (Vote for 3)

C – 2; G – 5; H - 6; I – 7; J – 0; K – 3.

Narrow the discussion to the top 3 vote getters here: G, H, and I.

Steve argues against G, as the rule is impossible to implement for dispatching calls, access-to-subprogram and the like. Depending on the exact rules, a dispatching call and a statically bound call might get different checking. He thinks that is unacceptable.

```
type T...
function F (P : T; ...)

type T2 is new T ...;

-- Inside a primitive of T:
XC : T'Class;
XS : T2;
```

```
F (XC); -- Dispatching call. The target routine doesn't know if is this is inner or outer.
F (XS); -- An "inner" call. Calling T2 is not checked.
```

Erhard says that calls from "inside" don't need the checks. That would allow holes in the checking for dispatching calls that dispatch into some other implementation. It also would seem to either require two sets of tags, as the checking has to be done in the body or a wrapper, or the distributed overhead of passing a flag to every primitive subprogram to determine whether to make the checks.

Straw poll: "inner" calls [G] compared to suppress on some **in** parameters [H/I].

G – 1; H/I – 7; abstain – 3

We turn to choosing between H and I. Erhard says that he thinks it is too weird to make function or procedure change the checking.

Geert suggests that we ought make this dependent on limited. That is, check **in** parameters of limited types in procedures, otherwise not.

Tucker notes that treating functions and procedures differently is similar to the rules for protected types. Everybody *loves* those (sarcasm set to maximum).

Geert note that procedures are unusual to use in invariant expressions. They can't be used in the top-level expression (that can only be a function), and it takes work to make trouble.

Ed argues for simplicity. A slightly larger hole is not a huge deal as we're never going to plug all the holes.

Tucker suggests that changing an **in** parameter (to make the invariant false) is a bounded error; either make the check or not. Steve worries about the problems that prevented us from suppressing assertions. We'll need to see the rule to see if there is a problem or not.

Action item to Tucker: write up a bounded error for **in** parameters that are changed to destroy the invariant.

A final straw poll: check on procedure – 3; never check **in** parameters – 8; abstain – 1.

On Friday, Tucker says that he thinks it would be better for this to be an Implementation Permission rather than a Bounded Error. There is a permission to check **in** parameters so long as they aren't called from the same invariant. This seems OK.

[Editor's note: It doesn't appear that we ever assigned updating the AI (wording, summary, and discussion) to anyone. I've assigned the entire thing to Tucker (as he is doing most of the work anyway – he surely needs to create some discussion to back up the Implementation Permission – and this was an editor-created AI originally, so there is no author to give it back to).]

AI12-0045-1/01 Pre- and Postcondition are allowed on generic subprograms

Tucker asks if you can put a precondition on a subprogram that is completed with an instantiation. He is reminded that you can't complete a subprogram with an instantiation, so it would be hard for this case to occur.

Bob points out that no one is asking for this on subprograms in generic package instances, which are far more common than generic subprogram instances. He thinks that the problem is therefore dubious.

Randy notes that allowing them only on generic subprogram instances would lock one into using a generic subprogram as it could not be changed into a generic package or access-to-subprogram parameter. This could cause a maintenance problem and unintentional incentive to use one form over other supposedly equivalent forms.

Tucker suggests that we add a note that you can use renaming-as-body to get this if needed (put the precondition on the specification, then use a renames-as-body to complete it).

The wording is messy, the wordy part is first. Perhaps:

> For a generic subprogram, entry, or subprogram other than the instance of a generic subprogram, following language-defined aspects may be specified with an aspect_specification:

This puts the most important part last, no one thinks that's a good idea. It is suggested that we don't have to say what the instance is of:

> For a subprogram other than an instance, a generic subprogram, or entry, the following language-defined aspects may be specified with an aspect_specification:

How about using "noninstance" instead? That gives:

> For a noninstance subprogram, a generic subprogram, or an entry, the following language-defined aspects may be specified with an aspect_specification:

Gary says (after checking) that there is several uses of “noninstance” in the Standard, so this is OK.

Approve AI with changes: 11-0-0.

AI12-0046-1/01 Enforcing legality for anonymous access components in record aggregates

In the question, modify the declaration of type T2: “type T2 is new {T1}[T2]”

Remove the sharps from the labels “#1)”.

Approve AI with changes: 8-0-2.

AI12-0047-1/01 Generalized iterators and finalization of the associated object

For question (1), change the question to say “this should be illegal, right? (Yes.)”.

Use Randy's wording (dropping the extra “a variable”) from the last e-mail to solve this. But is that enough?

Turning to question (2), the finalization occurs too soon. How do we fix? Currently, all “sinks” are scalar (case *selecting*_expression, condition, etc.).

Steve suggests changing 7.6.1(3/3) by adding “scalar” or “elementary” so that it only applies to such types. Tucker says it should be scalar, so change “an expression” to “a scalar expression”. This needs an AARM note to explain. Tucker now changes his mind, says it should be “elementary”, because you've made a copy and are done with it, while with composite, we still need the value. (It really should be any time that you have “by-copy”, but that's implementation-defined and we don't want this semantics to be implementation-defined.) We're still scared of “access”, so we stick with “scalar”.

Tucker would like to distribute “scalar” twice (range doesn't need it).

Steve will write an AARM note and fixup the discussion to match the wording agreed to above.

Approve intent: 9-0-1.

AI12-0047-1/02 Generalized iterators and finalization of the associated object

On Saturday, Steve presents his AARM note:

Add after 7.6.1(3.c/2) [continuing the existing AARM note]:

> The above definition states that only scalar, as opposed to all, expressions and function calls occurring immediately within a compound statement are masters. This distinction only makes a difference in the case of a loop statement with an iterator_specification because other compound statements (e.g., case statements) can only immediately enclose scalar expressions. When iterating over the result of a function

> call, we want that function result to be finalized at the end of the loop statement, as opposed to finalizing it before executing the first iteration of the loop. This rule accomplishes that.

Change:

"...this distinction only {matters}[makes a difference]..."

In the AI, !question, 2, lower case "for" in the example text.

Add after 5.5.2(6/3):

> The *iterator*_name or *iterable*_name of a iterator_specification shall not be a subcomponent that depends on discriminants of an object whose nominal subtype is unconstrained, unless the object is known to be constrained.

Tucker suggests replacing "be" with "denote", as this is a name, not an entity. Gary notes "a iterator_specification" (should be "an").

Approve AI with changes: 11-0-0.

AI12-0048-1/01 Default behavior of tasks on a multiprocessor with a specified dispatching policy

Tucker suggests using singular in the wording:

> If a task belongs to the system dispatching domain, unless it has been assigned to a specific CPU, it can execute on all CPUs that belong to the system dispatching domain.

Geert claims that this does not match with the behavior of Ravenscar. Randy digs up a Ravenscar paragraph D.13(8/3).

Tucker points out that there should not be additional rules applied to a profile. These have to be part of some other pragma or policy; a profile is only a selection of other pragmas (see 13.12(13/3)).

We should ask IRTAW to define a pragma or policy to encapsulate D.13.1(8/3-9/3). Randy will create an AI for that purpose and it to send to Alan and Tullio for presenting to the meeting. [That is now AI12-0055-1 – Editor.]

"Unless specified otherwise, ..." gets rid of the conflict. So the wording becomes:

> Unless specified otherwise, if a task belongs to the system dispatching domain, unless it has been assigned to a specific CPU, it can execute on any CPU that belongs to the system dispatching domain.

Another improvement to the wording:

> Unless the assignment of task is specified otherwise, if it belongs to the system dispatching domain, it can execute on any CPU that belongs to the system dispatching domain.

Still wordsmithing. Now:

> Any task that belongs to the system dispatching domain can execute on any CPU within that domain, unless the assignment of the task has been specified.

Approve AI with changes: 8-0-2.

AI12-0049-1/01 Invariants need to be checked on the initialization of deferred constants

Tucker sent some better wording this morning. He suggests:

Add a new bullet after 7.3.2(10/3):

- After successful explicit initialization of the completion of a deferred constant with a part of type *T*, occurring immediately within immediate scope of the full view of type *T*, the check is performed on the part(s) of type *T*;

Steve says the this doesn't cover a deferred constant declared in a visible child package of the package that defines *T*.

- After successful explicit initialization of the completion of a deferred constant with a part of type *T* that is inside the immediate scope for the full view of *T*, and the deferred constant is visible outside of the immediate scope of *T*, the check is performed on the part(s) of type *T*;

Tucker will wordsmith this.

On Friday, we consider the wording Tucker has sent:

- After successful explicit initialization of the completion of a deferred constant with a part of type *T*, if the completion is inside the immediate scope of the full view of *T*, and the deferred constant is visible outside the immediate scope of *T*, the check is performed on the part(s) of type *T*;

Approve AI with changes: 9-0-1.

AI12-0050-1/01 Conformance of quantified expressions

The quantified_expressions are different declarations, so we need to take that into account. The same identifier is required for both expreissions. Some says this is obvious; of course the intent is obvious, but the rules don't match the intent. Not every implementer and user has the same idea of obvious as we do, so we need to be explicit. Especially when it is additional work for implementations.

Steve will take this and create wording.

Approve intent: 11-0-0.

AI12-0051-1/01 The Priority aspect can be specified with Attach_Handler

The summary is confusing. Replace it by the following.

“The Priority aspect can specify the priority of an interrupt handler.”

Change the 12 to “...” in the example in the !question. It seems relevant but it is not.

In the wording, delete the extra word from “aspect pragma”.

Approve AI with changes: 10-0-0.

AI12-0052-1/01 Implicit objects are considered overlapping

Add a missing word in the summary: “language-defined {units}”.

There is a typo in the third line of the question: “the situation is murkier when on{e} of...”, and on the next line: “...rules do[es]n't...”.

In the summary, “implicit objects” should be “implicitly referenced objects”.

In the A.10.3(21) wording, “...the subprogram [should be]{is} considered to have a...”

Tucker would like to avoid a special case for reading directories and environment variables; he thinks that they should always work without extra complication. The problem only occurs for multiple writes. Randy argues that it doesn't matter, you'd still have to have a lock for reading if you need one for writing.

After discussion, we decide directories and environment variables aren't used enough to avoid the locking. We should require locking if needed on these packages (and it's quite possible that the underlying OS provides the needed locking). So we'll delete all of the text on these.

There should be some sort of AARM mention of this. Tucker suggests putting it in A(3). He thinks a statement that the rule of A(3) includes packages with global state, like environment variables, would be enough.

We should put all of the text (including the Text_IO place) into A(3).

The AARM note for Text_IO can stay where it is, pointing at the implreq in Annex A.

Approve intent: 7-0-2.

AI12-0053-1/01 Predicate failure raises Constraint_Error

[Editor's note: This AI number was assigned after the meeting.]

Randy suggests adding a second example to the discussion,

```
type Enum is (A, B, C, D);
subtype Sub is Enum range A .. C;
```

then add a new literal and use a static predicate so the subtype contains the same items as before:

```
type Enum is (A, B, E, C, D);
subtype Sub is Enum
   with Static_Predicate => Sub in A | B | C;
```

Here, it is weird to change the exception when changing from a constraint to a predicate.

Erhard wonders about the fact that Suppress is not effective on the latter. Suppress is based on checks, not Constraint_Error.

Suppress(All_Checks) is allowed to ignore assertions, so a “friendly compiler” could turn off the checks. Geert notes that all existing Ada 2012 compilers don't suppress assertions for "all_checks".

Simplify the wording for 4.6(57/3):

> If an Accessibility_Check fails, Program_Error is raised. [If a predicate check fails, Assertions.Assertion_Error is raised.] Any other check {including a predicate check} associated with a conversion raises Constraint_Error if it fails.

Someone suggests that predicate checks be moved to Suppress. We don't want to introduce erroneous execution when these are removed. (We already discussed this in detail during the Texas meeting.)

Randy and Geert note that range checks shouldn't have been erroneous, either, and we surely wouldn't want to expand that. (We have the mechanism of invalid values to handle cases of missed range checks, erroneousness is overkill.)

Tucker notes that predicates can't be erased from the program as they participate in membership tests (and 'Valid) even when "ignored".

Most likely, analysis tools for Ada 2012 will support static predicate analysis in a similar way to range checks, and safety-critical users will either ban dynamic predicates or only use them in the testing phase.

Writing handlers for predicate failure (or range check failure!) is unusual, so the run-time incompatibility is unlikely. That means that the exception raised isn't that important.

Straw poll on this idea: Make the change: 6, Leave it alone: 1; Abstain: 4

Some feel we need to decide on the other Duff AI (now AI12-0054-1) to decide. We discuss that for a while, then return to this AI.

Revote the straw poll: 5 – constraint; 4 – assertion; 2 – on the fence

Probably not enough support to continue. So forget this idea.

No action: 10-0-1.

AI12-0054-1/01 A raise_expression does not cause membership failure

[Editor's note: This AI number was assigned after the meeting.]

Steve explains the problem. Imagine we have the following predicate:

```
subtype S is Integer
   with Dynamic_Predicate => Is_Round(S) or else raise Program_Error;
```

Now consider:

```
if Obj in S then
```

This would raise Program_Error if `Is_Round(Obj) = False`.

It is pointed out that 'Valid needs to work the same as membership.

Tucker would like to restrict this special semantics to raise_expressions that have a Boolean type. That adds to the complexity of an already not very intuitive rule. It also would make it harder to use case_expressions in predicates, as `others => raise Program_Error` is fairly common. Tucker drops this idea.

Steve says that there is a problem with this rule as proposed. Consider:

```
subtype S is T with
   Dynamic_Predicate =>
      ... and
      ... and
      ... and not (Is_Bad(S) and then raise E);
```

Randy thought that the idea was that a raise_expression appearing here causes the entire predicate expression to return False. As opposed to the raise_expression returning False.

A call to a function in such an expression still can raise an exception; trying to handle that is likely to cover a bug. It also costs execution time as handlers would be needed; a rule like this is syntactic, only the result is modified.

So “the predicate yields false” rather than the raise_expression itself.

It is annoying that wrapping a predicate in a function would break memberships, but there doesn't seem to be a good solution to that.

Proposed changes: If you evaluate a raise expression in a predicate expression used in a membership or 'Valid, the entire predicate expression returns False.

Reword the summary.

Steve asks about default expressions in function calls that contain raise expressions; Randy piles on with <> in an aggregate. Bob will figure this out.

Approve intent: 8-0-3.

AI12-0054-1/02 A raise_expression does not cause membership failure

Bob rewrote this based on previous discussion.

In the !discussion, the following paragraph is confusing:

> Therefore, we change the rule so that "Is_Gnarly(S) or else raise Program_Error" is evaluated as "Is_Gnarly(S) or else False" -- but only for membership tests.

This true for this predicate expression, but not all predicate expressions. Is there some better way to explain this?

Randy complains that “contains” isn't clearly static.

We look at the wording change for 11.3(4/2):

> There is one exception to the above rule: For an individual membership test (see 4.5.2) of the form "X in S", where S has a predicate that contains a raise_expression, if that raise_expression is evaluated, then the exception is not raised; instead, the individual membership test yields False. This includes raise_expressions contained within the default_expression for a formal parameter, but not raise_expressions within the default_expression for a component.

"contains" probably should be "within".

Bob will send new wording.

Erhard wonders if it would be better to handle all exceptions here (rather than the syntax-based top-level only rule contemplated). The counter-argument is that it would hide bugs – most exceptions in this case are completely unexpected. It also would be much less efficient – handlers are more expensive than a text substitution.

Erhard is dubious that such a handler would hide bugs. Tucker points out that if you guard all calls with a membership test, you would never see the bug in a buggy membership.

```
if X in S then
   P (X); -- P has a parameter of subtype S.
else
   ...
```

If the predicate had a bug that caused it to raise an exception, and we turned all exceptions into False, P would never be called and the bug would never be seen. (Other than a mysterious omission of a call to P.)

The RM wording probably shouldn't include wording for components; there should be an AARM note. Add an example with default parameters to the AI.

Randy worries that he would convert a Mode_Error into an expression function, including the exception. And then the exception would get raised; the special behavior would disappear. He suggests that expression functions completed locally be expanded for the purposes of this rule. This idea attracts absolutely no support. [An alternative idea would be to expand inlined functions for the purposes of this rule. That's probably more attractive, but probably not enough. - Editor]

Erhard suggests that the model is that you don't hide exceptions in functions, period. Randy grumbles that this reduces the possibility of abstraction in predicates.

Bob will add some wording to explain this user model.

Keep alive: 10-0-0.

Tucker would like an e-mail ballot on this AI, once the wording is finished. He points out that predicates might be improperly written if it is not understood, and users shouldn't be using raise_expressions until this AI is decided one way or another. Randy will send Bob his notes ASAP.

Detailed Review of ASIS 99 SIs

SI99-0062-2/01 Forget the Semantic Subsystem

[Editor's note: This mainly is a general discussion about how to proceed on ASIS, and includes some discussion on this SI and SI99-0066-1 mixed in.]

Jean-Pierre says that the Semantic Subsystem is too different than the rest of ASIS. There is no interest in implementing it. But he still needs some of the capabilities that it has.

One problem is missing declarations for operators. To evaluate **use type**, he needs to know if an operator is primitive, but there is no declaration to query whether it is primitive.

Tucker suggests providing an operation that could query the call (name) to say whether the operator is primitive.

Jean-Pierre comments that one important thing that is missing is that there is no requirement to return the declaration that is visible. ASIS allows returning either (for declarations that have multiple parts), and thus processing private types and the like are much harder.

Getting back to the semantic subsystem itself. Clearly, we need to drop the semantic subsystem altogether. It doesn't make sense to make it an optional capability that we continue to have to maintain, as there doesn't appear to be any interest at all in implementing it.

So how do we unwind this? Randy notes that we changed a lot of the introduction to reflect the existence of the semantic subsystem. It's going to take a lot of work to scrub it back out.

Tucker volunteers to help with the unwinding of the introduction. [Silly man – Editor. :-)]

Jean-Pierre suggests asking Sergey Rybin at AdaCore to write up the queries that he already has implemented to solve some of these problems. He also has written some queries for Ada 2012 as well.

We clearly need to finish the Ada 2005 AI ASIS tasks. Randy wonders if we should just restart the old ASIS action item list, so that the people who did their ASIS homework last time don't have to do someone else's as well. This meets to approval (most likely from those who did that homework).

Bob is working on something called Ada Tree, Tucker thinks it might be helpful for people who find the existing ASIS interface frustrating. But Bob's not here yet (this was first thing Saturday morning), so we can't find out how appropriate it is.

Tucker will create an SI to eliminate the semantic subsystem from the Standard. This should get a new number, we'll delete SI99-0062-2. (The work items in it will go elsewhere.)

Reactivate the homework assignments for Ada 2012. Randy will find this list and circulate it (after figuring out who already did their work). Jean-Pierre notes that Sergey has already solved some of these problems and marked the source with special comments. Ed will extract these from the source and distribute them so people can use them to complete their homework. He needs to do that reasonably soon because everyone else will have to wait until he does that before doing their homework.

Jean-Pierre will take a look at the newer AIs that were not on his original homework list to see if there is anything missing.

Tucker will write wording for SI99-0066-1.

Tucker asks Bob if his Ada Tree package would be useful to add to the ASIS Standard (he's arrived since we last talked about this). Bob says he didn't design it to be part of the Standard. Bob is using ASIS.Extensions to implement it, it has a large “flat” enumeration; thus he doesn't think it would be appropriate for the Standard.

SI99-0065-1/01 Generic_Actual_Part and null procedure defaults

Jean-Pierre notes that this is the same issue as SI99-0051-1, but the answer is different. So he would like to use this solution and kill off SI99-0051-1.

Write text to undo the changes in SI99-0051-1.

Jean-Pierre will write the wording for this SI.

Approve intent: 7-0-2.

SI99-0065-1/02 Generic_Actual_Part and null procedure defaults

In the second paragraph of the discussion, change “...that [the] compiler{s} [was]{were} using an implicit[e]ly built null procedure.”

Gary wonders what an “implicit naming expression” is (this is used in the summary). Perhaps it is an ASIS technical term? Drop “naming” from that text.

The last sentence of the summary needs a period.

Write out “OTOH” in discussion; “GNAT” should be in all-caps.

Approve SI with changes: 6-0-4.