Minutes of the 55th ARG Meeting

11-13 June 2016

Pisa, Italy

Attendees: Steve Baird, John Barnes, Randy Brukardt, Alan Burns (not Saturday), Jeff Cousins, Brad Moore, Erhard Ploedereder, Jean-Pierre Rosen, Florian Schanda, Ed Schonberg, Tucker Taft, Tullio Vardanega.

Observers: Raphael Amiard, Stephen Michell.

Meeting Summary

The meeting convened on Saturday, 11 June 2016 at 09:10 hours and adjourned at 13:10 hours on Monday, 13 June 2016. The meeting was held in a conference room at the Grand Hotel Duomo in Pisa for Saturday and Sunday, and at TECIP on Monday. The meeting covered all of the normal AIs and most of the amendment AIs on the agenda.

AI Summary

The following AIs were approved:

AI12-0128-1/07 Exact size access to parts of composite atomic objects (9-0-3)
AI12-0182-1/02 Pre'Class and protected operations (9-0-1)
AI12-0194-1/01 Language-defined aspects and entry bodies (11-0-0)

The following AIs were approved with editorial changes:

AI12-0002-1/02 RCI units should not allow types with user-defined stream attributes (11-0-0)
AI12-0125-3/03 Add @ as a shorthand for the LHS of an assignment (10-0-2)
AI12-0140-1/03 Access to unconstrained partial view when full view is constrained (9-0-2)
AI12-0170-1/03 Abstract subprogram calls in class-wide precondition expressions (9-0-2)
AI12-0180-1/01 Using protected subprograms and entries within an invariant (10-0-1)
AI12-0181-1/02 Self-referencing representation aspects (9-0-2)
AI12-0184-1/01 Long Long C Data Types (11-0-1)
AI12-0185-1/01 Resolution of postcondition-specific attributes (11-0-0)
AI12-0192-1/01 "requires late initialization" and protected types (8-0-3)
AI12-0195-1/01 Inheriting body but overriding precondition or postcondition (6-0-5)
AI12-0198-1/01 Potentially unevaluated components of array aggregates (11-0-0)

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

AI12-0004-1/01 Normalization and allowed characters for identifiers (11-0-1)
AI12-0064-2/02 Nonblocking subprograms (11-0-1)
AI12-0127-1/03 Partial aggregate notation (12-0-0)
AI12-0179-1/01 Failure of postconditions of language-defined units (9-1-1)
AI12-0186-1/01 Profile freezing for the Access attributes (10-0-1)
AI12-0187-1/01 Stable properties of abstract data types (9-0-2)
AI12-0189-1/02 Loop-body as anonymous procedure (9-1-2)
AI12-0191-1/01 Clarify "part" for type invariants (10-0-0)
AI12-0193-1/01 Postcondition failure for a task entry (9-0-2)
AI12-0196-1/01 Concurrent access to Ada container libraries (10-0-2)

The following AIs were discussed and assigned to an editor:

AI12-0079-1/03 Global-in and global-out annotations
AI12-0111-1/03 Tampering considered too expensive
AI12-0139-1/02 Thread-safe Ada libraries
AI12-0171-1/00 Ambiguity in Synchronous_Task_Control semantics
AI12-0190-1/01 Lambda functions
AI12-0197-1/01 Generator functions
AI12-0197-2/02 Passive Tasks

The following AIs were discussed and voted No Action:

AI12-0125-1/05 Add Object'Inc and 'Dec (12-0-0)
AI12-0125-2/02 Add :+, :-, :*, :/, … (10-0-2)

Detailed Minutes

Apologies

Gary Dismukes and Bob Duff apologize for not being able to attend.

Previous Meeting Minutes

Brad notes that he had a homework item (AI12-0163-1), that was actually finished. That should be removed. Approve minutes with changes: 11-0-0.

ARG Procedures Update

Randy gives an overview of the changes. 17% was selected in an ad-hoc manner; it provided results that felt right, avoided anomalies that occurred for rounder values, and does not provide an integer result for any reasonable number of voters (so there is no doubt as how to round).

Approve changes to procedures: 11-0-0.

Date and Venue of the Next Meeting

Our next meeting will be in Pittsburgh, following the HILT conference. WG 9 will be the morning of Saturday, October 8. The ARG meeting will start in the afternoon of October 8th and end in the early afternoon on Monday, October 10th.

Will there be meeting rooms for those dates? Tucker will check. Tucker will also look at hotel rooms for the meetings. He will try to have that done in a few weeks.

The following meeting most likely will be associated with the 2017 Ada-Europe conference, June 12-16, 2017, in Vienna, Austria. (We didn't discuss when the ARG and WG 9 meetings would be.)

Thanks

Thanks to the editor (Randy Brukardt) for taking the minutes.

Thanks to the Rapporteur (Jeff Cousins) for running the meeting.

Thanks to Ada-Europe for accommodations and arrangements.

Unfinished Action Items

AI12-0016-1 and AI12-0020-1 will get done when Steve gets around to them.

AI12-0075-1 was supposed to have some research done. Steve says that in all probability, there would be no incompatibility for the simple rule. Randy notes that the simple proposal still leaves another possible incompatibility, which also should be researched. We also wanted to know whether Bob's aggressive option is too incompatible. Randy notes that he'd be more inclined to support the proposal if we used Bob's option, thus we need some data on its acceptability. Someone notes that expression functions are new, so we might be more accepting of an incompatibility there. The AI was left with Steve.

AI12-0112-1, just assign to Randy (he has included some of this work in his budget for 2016). AdaCore will review.

AI12-0164-1 was assigned to Steve Michell to get information from IRTAW. He apologizes for not making a report, he will provide a report today (which he did - Editor). A synopsis is that a new profile will use this feature (IRTAW does not want to change Ravenscar itself.) AdaCore is looking at a possible definition of a new profile.

AI12-0119-1, parallel loops, was assigned to Brad. He expects to have something for Pittsburgh; there is a session at Ada-Europe on this topic.

Erhard sent a draft of AI12-0139-1, on June 9 [after the deadline - Editor], but it did not get posted to the list.

Erhard would like to remove the other item; he cannot figure out how to cut the example to something of reasonable size. We ask him what the problem seems to be. His understanding was that it was related to GNAT inlining and "some data structure which is allocated which is unneeded". We ask Erhard which operations are being used from the containers. (He thinks it was on inserting.) Florian will try examples using those operations with a lot of (random) data.

AI12-0058-1 (Fortran interface updates): Tucker would like to find someone else to do this. Brad will take this, Ed will stay on as a secondary.

AI12-0017-1, exception contracts: Florian plans to work on it (someday).

AI12-0064-1: Tucker would move to Randy's AI12-0064-2, so he didn't update this version.

Tucker will do something for “parallel reads” for Pittsburgh.

Current Action Items

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

Raphael Amiard:

- AI12-0111-1 (prototype stable vectors)
- AI12-0197-1 [Editor's note: it doesn't make sense to work on both this and the alternative – pick one.]
- AI12-0197-2 (assist Jean-Pierre Rosen; also provide better examples)

Steve Baird:

- AI12-0016-1
- AI12-0020-1
- AI12-0075-1
- AI12-0179-1
- AI12-0191-1 (assist Tucker Taft)
- AI12-0193-1
- AI12-0197-1 (alternative syntax proposal)
- New AI to improve the wording of A(3/4) for concurrent calls (see discussion of AI12-0196-1, assistance from Tucker Taft). [Assigned AI12-0200-1 after the meeting – Editor.]
- New AI for Container aggregates (help Florian Schanda; see discussion of AI12-0189-1)

Randy Brukardt:

- AI12-0004-1
- AI12-0064-2 (with assistance from Tucker Taft)
- AI12-0112-1
- AI12-0187-1

Editorial changes only:

- AI12-0002-1
- AI12-0125-3
- AI12-0140-1
- AI12-0170-1
- AI12-0180-1
- AI12-0181-1
- AI12-0184-1
- AI12-0185-1
- AI12-0192-1
- AI12-0195-1
- AI12-0198-1

Bob Duff:

- AI to define suppress check name Tampering_Check (see discussion of AI12-0111-1 in meeting #54)

Brad Moore:

- AI12-0058-1 (lead to produce wording, with Ed Schonberg and Van Snyder)
- AI12-0119-1

- AI12-0196-1

Erhard Ploedereder:

- AI12-0139-1 (protected containers)
- Provide information about which operations are involved when Ada containers are too slow compared to other containers (for AI12-0111-1). Give this to Florian. (See discussion under Unfinished Action Items.)

Jean-Pierre Rosen:

- AI12-0197-2 (with help from Raphael Amiard)

Florian Schanda:

- AI12-0017-1
- AI12-0190-1 (with Tucker Taft)
- Provide better examples for AI12-0197-1
- New AI for container aggregates (with help from Steve Baird; see discussion of AI12-0189-1)
- Test container operations noted by Erhard using a lot of data. (See discussion under Unfinished Action Items.)

Ed Schonberg:

- AI12-0058-1 (with Van Snyder and Brad Moore)

Van Snyder:

- AI12-0058-1 (with Ed Schonberg and Brad Moore)

Tucker Taft:

- AI12-0079-1
- AI12-0111-1 (AI update, coordinate with Raphael Amiard)
- AI12-0171-1
- AI12-0186-1
- AI12-0189-1
- AI12-0190-1 (with Florian Schanda)
- AI12-0191-1 (with help from Steve Baird)
- AI12-0199-1 (split from AI12-0170-1, see discussion of that AI).
- Parallel reads for existing containers (see discussion of AI12-0139-1 in Madrid, possibly related to AI12-0111-1)
- Assist Randy Brukardt with AI12-0064-2
- Assist Steve Baird with new AI to improve the wording of A(3/4) for concurrent calls (see discussion of AI12-0196-1).

Detailed Review

The minutes cover detailed review of Ada 2012 AIs (AI12s). The AI12s 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 working Ada 202x AARM, the number refers to the text in draft 8 of the Ada 202x AARM. Paragraph numbers in other drafts may vary. Other paragraph numbers come from the final consolidated Ada 2012 AARM; again the paragraph numbers in the many drafts may vary.

Detailed Review of Ada 2012 AIs

AI12-0002-1/02 RCI units should not allow types with user-defined stream attributes

“stream-oriented attribute” in the summary, subject.

The subject and summary should be: “RCI units do not allow specification of user-defined stream-oriented attributes”

Approve AI with change: 9-0-2.

AI12-0004-1/01 Normalization and allowed characters for identifiers

Randy explains the problem. He believes that he misclassified this AI originally; it should be a Binding Interpretation to correct the reported portability problem. Additionally, we definitely don't want the set of legal identifiers to be implementation-defined (and it is, currently).

He continues to say that he has come to believe that changing the identifier character classes is too risky at this point. We probably should have used those in the first place, but we can't justify any incompatibility in identifiers (and we've already had issues with that with the Ada 2005 rules). So the answer to the second question should be “No.”, after dropping “consider” from it. (Yes, we considered it, but no we're not going to change it – that's too confusing of a question.)

Florian follows up with a confused discussion about problems with Unicode and use of it in Javascript. We point out that we're stuck with it; supporting it was a directive from SC 22 – we had no choice in the matter.

Randy suggests that the intent be that identifiers are restricted to characters that are in normalization form KC. The rest of the program text allows any characters. Nothing will be implementation-defined. Change this to a Binding Interpretation.

Approve intent of AI: 11-0-1.

AI12-0064-2/02 Nonblocking subprograms

S'Nonblocking is True in the generic body, and in the generic spec, we should assume it is False. (Assume-the-worst in the body; assume-the-best in the spec). (Recheck the spec. could happen in an expression function, or in an instance in the spec.)

We should allow to specification of the aspect on the formal, which would affect generic actual matching. We need to be able to call a formal from inside of a protected type that has Nonblocking => True.

Randy asks about a subprogram of a limited interface being implemented by an entry. That cannot have nonblocking => True (it can be overridden by an entry). So the bullet about synchronized interfaces is unneeded because the same applies there.

Jeff wonders why this is in protected operations. Because that's where “nonblocking” is defined.

Tucker suggests that we move it to 9.5, along with most of the bounded-error section of 9.5.1.

Approve intent of AI: 11-0-1.

AI12-0079-1/03 Global-in and global-out annotations

Steve wonder if this change would have an adverse effect on SPARK users? Florian doesn't think so, so long as this is worded so as to allow SPARK to use its state abstractions.

Ed wonders if these are compile-time enforced? Yes. Does SPARK do that? Yes, but not in the front end.

Steve wonders where storage pools go; they appear on (as part of) the access type.

Florian notes that SPARK allows constants with non-static inputs to appear in these, that would need to be allowed.

Florian wants the default on a package to not be called Global; Global on the package means that these are the globals for the elaboration.

Florian notes that SPARK uses “Input”, “Output”, “In_Out” because they couldn't use reserved words. He'd rather use “**in**”, “**out**”, “**in out**”. He'd also like to make it more like a parameter list.

Florian also notes that SPARK allows “proof” as a fourth mode. That would need to be allowed. Someone asks what it is for. It's mainly useful for ghost code.

Tucker wonders why Florian hates Input and the others. He doesn't like the capitalization, and it's the same idea. Randy notes that the unfinished Ada 2005 proposal used “**in**”, “**out**”, etc. He thought that this needed all new syntax, it doesn't fit with anything. John notes that his initial feeling was “yuck”, and still feels that way.

Steve dislikes using & here. SPARK could get two different answers if the order is different.

Someone wonders what the difference between **out** and **in out** in one of these annotations. These aren't the same, as **out** promises that no reading of the object (and it is fully initialized); **in out** makes no promises.

Florian notes that the order of combination can matter. He gives an example:

```
procedure P1 with Global => out Foo;
procedure P2 with Global => in out Foo;

procedure P3A is
begin
   P1;
   P2;
end P3A;
-- Global => out Foo;

procedure P3B is
begin
   P2;
   P1;
end P3B;
-- Global => in out Foo;
```

In P3A, P1 guarantees to only write Foo, and that the object is initialized. P2 then touches the previously initialized item. So the global mode can be **out**. For P3B, the first call makes no guarantees, so the mode has to be **in out**.

So unordered combination of annotations doesn't work.

Tucker thinks that we want to use rules that are similar to parameter passing.

Florian suggests that there is two combinators: a merge, and one that uses overwriting semantics. (Essentially they are ordered and unordered combinators.)

A case like P3A isn't likely to be common, usually one or both of these calls is in a dynamic conditional or an iteration; you'd have to assume the worst anyway. An Ada (as opposed to SPARK) does not require any flow analysis; we wouldn't want to change that. It seems OK to do a conservative merge, so P3A would be **in out** Foo.

Tucker used & as it means “combination”, it's not quite “and” or “or”.

Florian asks about initializing a pair of objects A and B in a package body P:

```
procedure Initialize_A with Global => in out P;
procedure Initialize_B with Global => in out P;
```

but

```
procedure Initialize_All with Global => out P;
```

Tucker says that **out** just means that you write it. **out** doesn't promise that everything is written. Florian says that the definition should be compatible with SPARK.

Florian asks about state in the private part, and state in the children. The Tucker rules have the effect of including private part, the body, the private and visible children when referring to the package name.

Is that too much? The body can see things that the specification cannot name (especially private children).

SPARK uses “state abstractions” to deal with this. You can have multiple of these, they hold some part of the body state, including child packages.

Florian suggests that the visible items be excepted from the package name. So private parts, bodies, private children. Visible children would be named separately.

Erhard wonders how the global represents a package that you never heard of; which is implementation specific. You don't want to name that explicitly. You could use the global attribute.

Randy wonders about the implementation-defined packages that implement, for example, Text_IO. How do we say that in the standard? It is suggested that global is implementation-defined in that case.

Steve wonders if **limited with** could be used to help when we have to make visible packages that are only used in the body.

Tucker should look at the **limited with**, look at improvements in the syntax. (**in**, **out**, **in out** – whatever is done is not a mixture.) Look at removing visible children, etc. And add permissions for SPARK. Back to Tucker.

Keep alive: 12-0-0.

AI12-0111-1/03 Tampering considered too expensive

When tampering with elements is removed, it is replaced by tampering with cursors (it's not removed outright). The AI description is confusing on that point.

Tucker explains the stable view of the container; he expects that the compiler could substitute the stable versions for iterators over the cursor. That's the most frequent use, thus the complication isn't a big deal.

Steve wonders why it is in a separate package, shouldn't it be nested in the package?

The operations that tamper with cursors aren't provided in the stable version.

Tucker thinks that Merge should be tampering with cursors; there is an AARM comment about that A.18.2(94.a/2). But that makes no sense, Merge talks about insert/deleting of elements. Tucker seems to be correct that Merge should be included in tampering with cursors.

We discuss various ways to structure this. Is it a child of a generic vs a nested package?

Tucker shows an example of how this would be used.

```
package Int_Vects is new Ada.Containers.Vectors (Integer, ...);

package My_Stable is new Int_Vects.Stable;

V : Int_Vects.Vector;

declare
   S : My_Stable.Vector (V'Access);
begin
   ...
end;
```

The stable vector would trigger the tampering flag on V; and stable vector objects would be controlled to clear this.

My_Stable.Cursor is defined; these can outlive the stable view and we need say what happens in this case.

We would like to see a real implementation to see if this will actually help. Raphael volunteers to prototype this and report back.

Steve wonders if a stable view could have a default of null; that would help making a ragged array. (The stable view would have to be limited.) That allows components that don't define the discriminant at the point of the type declaration.

Tucker later sent an e-mail suggesting that a stable container be limited, since := would almost always fail a discriminant check before the assignment could operate. And then the null case could be eliminated in favor of a co-extension.

```
SB : My_Stable.Vector (new Int_Vect.Vector (...));
```

The existing proposal would allow:

```
SV : My_Stable.Vector (null) := A & B;
```

but makes more sense to leave the discriminant out in this case:

```
SV : My_Stable.Vector := A & B;
```

Steve says that the consistent version without the **null** makes more sense.

Florian wonders if we should rethink the entire container model rather than trying to incrementally fix the this model.

Ed notes this would not help if there is dynamic use of containers. Tucker notes that many times that one builds a single structure, then operates on it for a long time. Jean-Pierre notes that ASIS works this way; Florian notes that a SPARK tree also works this way.

Raphael and Ed claim that cursors need to be controlled to make proper tampering checks. Randy disagrees that this could possibly be necessary. This is discussed for a while, and then taken off line. (Off-line, they determined that they were wrong; cursors do not need to be controlled.)

Open questions:

Should it be limited? Nested vs. generic child?

Straw poll: Nested: 5; child generic: 1; abstain: 6.

Straw poll: Limited: 5; Nonlimited: 0; abstain: 7.

Keep alive: 11-0-1.

Tucker will redraft the AI; Raphael will prototype an implementation.

AI12-0125-1/05 Add Object'Inc and 'Dec

With the approval of alternative 3, this AI is unneeded.

No Action: 12-0-0.

AI12-0125-2/02 Add :+, :-, :*, :/, …

With the approval of alternative 3, this AI is unneeded.

No Action: 10-0-2.

AI12-0125-3/03 Add @ as a shorthand for the LHS of an assignment

“causal” => “casual”.

Erhard would prefer “abbreviation” to “shorthand”.

Capitalize the words in the title of the clause.

Title of the clause should be “Target Name Symbol”

Drop “*T*” from the Name Resolution Rule. “The target name is a constant view ...”

Target_Name in the syntax should be in lower case.

Why did we not use a rename? Because it was illegal for discriminant-dependent component. This model is the correct semantics.

> If a target_name with nominal subtype *S* appears in the expression of an assignment statement *A*, then *A* is equivalent to a call on a local anonymous procedure with the actual parameter being the *variable_*name of *A*, where the local anonymous procedure has an **in out** parameter with unique name *P* of subtype *S*, with a body being *A* with the *variable_*name being replaced by *P*, and any target_names being replaced by the qualified expression *S*'(*P*).

Meta discussion on qualified expressions and constant views.

Add to the start @Redundant[The *variable_*name is evaluated only once.] In particular,

Drop that statement from the AARM Reason.

Florian thinks he has a nice example:

```
Something_Long(1).C := Clamp(L, @ + X, U);
```

Clamp reduces the value to the range L .. U. The point is that you can use this abbreviation in ways that you couldn't use :=+ or the like.

Approve AI with changes: 10-0-2.

AI12-0127-1/03 Partial aggregate notation

Erhard asks if one can change the discriminants with a delta aggregate? No, it got very complicated, and changing a discriminant usually requires giving the entire value.

Are there any limitations on the base expression? No the base expression can be limited, this still allows updating the nonlimited components.

The tag of the base expression is preserved (for a tagged type), the nominal subtype isn't used. Steve notes it is like 'Old.

Erhard complains that an array delta aggregate should include “single dimensional” in order to contrast with the multidimensional array” case. Steve notes that it has been suggested to give these guys individual syntax, and then the rules would use these names (and the resolution would be separated).

Trapazoid => Trapezoid in the problem.

The examples need to be based on types found earlier in the ARM. Some of the “awkward” examples should go in the AARM or just the AI.

The second example needs a semicolon. “non-limited” does not have a hyphen.

Overlapping components is allowed for arrays, the evaluated order is as written. There's no applicable index constraint, so **others** isn't allowed.

Erhard is concerned that an aggregate has an order of evaluation; this seems new. Tucker notes that extension aggregates have an order of evaluation (but not as strict).

Tucker thinks that the record delta aggregate should be also be required to be in the given order, so that all delta aggregates are consistent.

Florian explains the multi-dimension extension.

Tucker suggests that we just focus on the record and single dimensional array cases, because the multidimensional array is so ugly.

Straw poll:

For this AI, drop multidim completely: 5; Basic version of multidim: 3; Full version: 3.

Split the multidimensional case out into a separate AI. And make RM appropriate examples.

Approve intent: 12-0-0.

AI12-0128-1/07 Exact size access to parts of composite atomic objects

Approve AI: 9-0-3.

AI12-0139-1/02 Thread-safe Ada libraries

Erhard prototyped one library in a safe version (Hashed_Maps).

Erhard drops tampering checks from the safe version, that would deadlock if it fails. (It blocks until the read lock is freed, that could never happen in a tampering situation.) Thus his version is “safe”. (He didn't promise “friendly”).

Tucker would like “Safe” changed to “Synchonized” or something like that. “Safe” is over-promising.

Keep alive: 12-0-0.

AI12-0140-1/03 Access to unconstrained partial view when full view is constrained

This sort of repeats 3.10.1(2.2-2.6/3), and that is dangerous. Perhaps we could get rid of the duplicate text. But we have to be careful, because 3.10.1(2.7/3) uses the bullets.

In the question, the error message and the explanation are screwy. It should be talk about discriminated and undiscriminated rather than constrained. The error message was just a compiler being confused about a static matching failure that the author didn't anticipate.

Tucker muses that he might be able to get the wording to work without any exception at all, as 3.10.1 says that the subtype is still incomplete in the appropriate cases. His !discussion should explain this, if that's true.

He also suggests saying “characteristics and constraints”. So we have:

> The characteristics and constraints of the designated subtype of an access type follow a somewhat different rule. The view of the designated subtype of (a view of) an access type at a given place is determined by the view of the designated subtype that is visible at that place, rather than the view at the place where the access type is declared.

Randy will add a bit of discussion to explain that 3.10.1(2.7/3) means that this does not override incomplete views; one uses the 3.10.1 rules to determine if a subtype is incomplete.

Approve AI with changes: 9-0-2.

AI12-0170-1/03 Abstract subprogram calls in class-wide precondition expressions

Tucker explains the changes.

We need to figure out whether any other changes are needed in 7.3.2. There is a dynamic rule “corresponding operations”. But is the recheck defined? Make an AI specifically for invariants, move all of the invariant rules there. [This AI was assigned number AI12-0199-1 after the meeting.]

Tucker will take the new AI.

Approve AI with changes: 9-0-2.

AI12-0171-1/00 Ambiguity in Synchronous_Task_Control semantics

We want wording that multiple tasks can't race on a Suspension_Object. We don't want to pay an extra expense to prevent problems from this race condition. (IRTAW says that it is fine that this doesn't work.) It's clear that the current wording isn't specific enough.

Tucker will propose some wording later today. (But he didn't, and we did not discuss this AI again during this meeting.)

AI12-0179-1/01 Failure of postconditions of language-defined units

Tucker and Erhard would prefer that this says that the postcondition shall not fail. We don't want this to change depending on the Assertion_Policy.

Put it after 1.1.3(15), because it doesn't have anything to do with Specialized Needs Annexes.

“When a postcondition or type invariant is defined for a language-defined unit, the checks associated with such postconditions or type invariants shall not fail in a conforming implementation.”

The AARM notes are OK.

There is a long argument about “failed check” in the case of the failure of a language-defined check vs. an explicit raise of an exception.

Jean-Pierre suggests long-winded wording that the editor did not record. Randy suggests that a To Be Honest note would be sufficient to define this. John worries that we are trying to get rid of English in the Standard by writing more English.

We will define “failed check” better in 6.1.1 or 11.4.2 or something. Steve Baird will take the AI.

Approve intent of AI: 9-1-1. John objects; he thinks that we are writing more text than we could save.

AI12-0180-1/01 Using protected subprograms and entries within an invariant

Summary: “with” should be “within”. In question: “hoever”.

Approve AI with changes: 10-0-1.

AI12-0181-1/02 Self-referencing representation aspects

Tucker does not like the first paragraph of the summary. Replace it by a copy of the new paragraph after 13.1(9.1/4), simplified by dropping “or name” and “representation or operational”.

Approve AI with changes: 9-0-2.

AI12-0182-1/02 Pre'Class and protected operations

Approve AI: 9-0-1.

AI12-0184-1/01 Long Long C Data Types

Randy worries that we're requiring 64-bit support on all machines except the tiniest.

Steve notes that any compiler can add these if they want, B.3(62).

Tucker suggests just making it Implementation Advice that an implementation support these if their C compiler does. This uses the permission of B.3(62).

> An implementation should provide unsigned_long_long and long_long as 64-bit types if the C implementation supports unsigned long long and long long as 64-bit types.

Update the discussion, proposal, and summary.

Approve AI with changes: 11-0-1.

AI12-0185-1/01 Resolution of postcondition-specific attributes

Typos: “prreviously” in !wording. In question, “(in a type conversion{)}.” The first open paren in that sentence should be removed.

“The type and nominal subtype of X'Old is ...”

Approve AI with changes: 11-0-0.

AI12-0186-1/01 Profile freezing for the Access attributes

We discuss 13.14(15) extensively. Tucker claims that the authors “forgot” that subtype_mark includes a name. That seems to be the only sensible understanding, but it is amazing.

Exclude stuff, not make it inclusive. Exclude designated subtypes and designated profiles. Tucker will take the AI, and figure out the wording.

Approve intent of AI: 10-0-1.

AI12-0187-1/01 Stable properties of abstract data types

Tucker wonders if this is only for private types. Randy and Florian both say it makes sense for record variants as well.

Steve Baird wonders if this interacts somehow with class-wide postconditions. Tucker thinks that it should.

Florian likes the idea of the feature.

Typo in the !discussion: "is Is_Open" -> "if Is_Open".

Florian wonders how one turns off if there isn't a clear way to describe this in the postcondition. Tucker suggests writing something like `Is_Open in Boolean` (with “Boolean” here being replaced by the return type of the function).

Is the all-or-nothing rule the correct thing? Randy notes that you can't always call each of them (as in the case of Mode), so it seems necessary to have some sort of grouping.

Florian suggests that we could add a Stable_Priorities aspect for a subprogram, which would override the list from the original type.

Tucker suggests that we only turn off the one that is mentioned in the postcondition. And an override for an individual subprogram would be required. Florian agrees.

The suggestion is to have a **not** property to suppress them; or a positive list. But one can't mix these (only **not**s or only positive). The syntax would be **not** A, **not** B.

We could define a package-level version. Randy worries that we could load up this proposal with too many bells-and-whistles until it falls over. We'll look at a package version later as a separate AI.

Approve intent of AI: 9-0-2.

AI12-0189-1/02 Loop-body as anonymous procedure

This is related to a bunch of other AIs.

This quickly devolves into a discussion about meta issues about lambdas, aggregates of containers, and even how other languages map these things.

We would like a new AI for aggregates of containers, possibly using similar iterators. Florian will take, and Steve will help him polish it.

We will put AI12-0009-1 & AI12-0188-1 on hold. This AI provides an alternative, potentially better, way of addressing the issues addressed by those AIs. For now, just defer them from this meeting: 11-0-0.

We finally turn to discussing this proposal itself.

Tucker wonders if we should allow just giving the formal parameter names, or do we need the complete profile. And the second question is how to deal with exits (as well as gotos and returns).

Steve notes that we would not want to do anything that would prevent dealing with exits using a flag or whatever, rather than exceptions.

Steve continues to comment that ATC seems like a scary option; we don't want to bring any of those problems into this. We shouldn't mention that.

Steve notes that a build-in-place function might be a problem; the return value would have to be staged somewhere.

Randy notes that an anonymous exception that cannot be handled by an **others** is a new concept. Tucker claims that how ATC/abort gets implemented. Not in Randy's compiler!

Erhard notes that "**others**" is used for last wishes. The private exception would not be handled, and that could prevent Iterate from cleaning up.

Tucker suggests that an alternative would be to the private exception idea would be to have an exception that we could handle but would have to be reraised. (Even if another exception happened.) Probably a better idea.

Hunger causes us to defer completing this discussion until tomorrow (Sunday).

Sunday morning, we continue discussing how to map **exit**, **goto**, **return**, and so on. The Iterate procedure needs a way to get control to clean up, so there needs to be some sort of exception involved.

Florian wonders why we don't have an extra terminate parameter to Iterate. The problem with that is that there is an unlimited number of possible goto and requeue targets possible. How could we map those?

Tucker says that we could just say that Iterate could be defined such that it has to propagate some known exception. If not, it is wrong, and then **exit** and the like do not work.

Tucker worries that we don't want to require that the Iterate procedure propagate all exceptions. Randy suggests that we give this exception a real name in some language-defined package. Then we can say that the Iterate procedure has to propagate this exception.

The alternative is to say the mechanism is undefined, but then it cannot be handled by the Iterate procedure for last wishes. Not having a clean-up mechanism seems worse.

Florian revisits his idea; since extra state to determine what to do is necessary, having a stop iterate parameter doesn't seem that bad. Jean-Pierre says that you should raise an exception to stop iterations (that's what the AARM suggests for the original container iterators).

Zero cost exceptions are quite expensive to be raised. (The raise is 50 times more expensive than the other way.) There is some concern that this would be too expensive.

Tucker says that the alternative is to say that no last wishes at all, and the only way to do so would be to use finalization.

Erhard suggests that the model is as-if the Iterate and Process procedures are in-line expanded. Then finalization happens when exit, goto, or return.

Steve insists on imagining accept statements in such a loop– since textually these are **not** nested. We'd have to make those illegal somehow (as they wouldn't be legal in the subprogram).

Approve intent of AI: 9-1-2. Erhard does not like the train of thought; because once we use access-to-procedure we then have issues with control structures. And then it looks like a much worse idea. Conceptual load is bigger than the value.

Tucker rebuts that this is an implementation concern. The user would not see (or care) about any of this. [Editor's note: That's not quite true if the user needs last wishes and is implementing a call-back iterator..]

Steve wonders what constitutes misuse of the access-to-procedure value; we'd hope that it couldn't get stored into a long-lived object. Another topic for Tucker to address.

AI12-0190-1/01 Lambda functions

Tucker says that"lambda_parameter_list would also allow a regular parameter list (that is you would have types there).

Raphael explains why "lambda functions" are important. One is a syntax-sugar for a nested subprogram. He also believes that it can make a program more expressive. In other languages, they often "capture" the parameters (essentially continuations).

Erhard notes that it can be used to reduce the number of parameters of a function.

We definitely don't want to do continuations.

Raphael shows an example of what we don't want to do:

```
function Make_Addr (I : Integer) return
   access function (J : Integer) return Integer is
begin
   return (lambda (X : Integer) X + I);
end Make_Addr;

Addr : access function (J : Integer) return Integer := Make_Addr (12);

Put_Line (Addr (23)'Image);
```

This would be nasty if Integer was replaced by a controlled type (or a task type).

Florian notes that lambda help writing stuff in specifications; you would avoid needing to write "ghost" expression functions.

Tucker notes that this example has the same problems as generate vis-a-vis accessibility. He would suggest treating this as an access discriminant.

Most languages, this is a copy (if at all possible). Even for local variables.

Erhard says that a simple version isn't interesting. People he knows are only interested in being able to return constructed lambdas.

Steve wonders if supporting a syntax sugar only version would do more harm than good. People would think that we are just faking.

Erhard thinks we could try to make this safe by the right restrictions, that would be a boon. (C++ just crashes if one does.)

Keep alive (original version): 2-4-6.

Keep alive (complex version): 6-1-5. (A complex version would support some sort of constructed lambda.)

Tucker will take another shot at this. Florian would like to help.

Procedural question: Votes on keep alive shouldn't fail if there aren't a lot of against (assuming that someone is willing to take AI); early in a proposal, there are a lot of abstentions. The new procedures don't make an exception for this case. [We didn't actually decide to do anything for this – Editor.]

AI12-0191-1/01 Clarify "part" for type invariants

A long discussion of whether the invariant would be checked on upcasts. ("Upcast" is a type conversion toward the root of a derivation tree.) We eventually determined that it is checked. There's no hole, but there is the somewhat odd possibility of the invariant failing on an upcast.

We then turn to understanding the note added by To Be Honest for AI12-0167-1.

Finally turn back to the question. The basic problem is that finalization uses "parts" determined dynamically (that is, parts in extensions not necessarily known to the compiler still get finalized), whereas type invariant checks (and probably other things) use "parts" determined statically (only parts that the compiler is aware of participate).

Either we add another To Be Honest, or define something like "statically known part". Tucker guesses that we probably want to define something like that; Steve suggests that "part" mean statically determined parts and then some other term mean dynamically parts.

Tucker volunteers to look at the entire standard to classify each use of "part" to determine whether it is meant statically or dynamically. Probably "component" and "subcomponent" as well. Steve volunteers to help find out what GNAT does on cases of "parts" that are only dynamically known for type invariant checks.

Jeff reports that there are 802 parts, 757 components, and 23 subcomponents in the Standard. Sounds like Tucker is in for lots of fun reading.

Approve intent of AI: 10-0-0.

AI12-0192-1/01 "requires late initialization" and protected types

Steve explains the original problem. Randy explains the unlikely case where legal, portable code becomes erroneous.

We talk a bit about this; Randy agrees that the case is very unlikely to occur. The problem is explaining that this case exists (not doing that would be essentially lying about the safety of this new version) without scaring users.

There are two ", or" in this wording, and we don't want these to be treated as one list. So the second part becomes a list of three bullets.

> A component of an object is said to *require late initialization* if:
> - it has an access discriminant value constrained by a per-object expression;
> - it has an initialization expression that includes a name denoting an access discriminant; or
> - it has an initialization expression that includes a reference to the current instance of the type either by name or implicitly as the target object of a call.

The incompatibility documentation needs to emphasize that this usually will fix bugs or not matter. The case in question is extremely unlikely.

Approve AI with changes: 8-0-3.

AI12-0193-1/01 Postcondition failure for a task entry

Subtype predicates probably are totally outside the action; they are only known to the caller.

Type invariants, however, belong to the entry, and should also be done inside. Randy notes that we don't have any rules that require type invariants should happen in a protected action or rendezvous. And there are rules on type conversions, and stuff. This looks very hard.

Therefore, Randy preferred that they all were outside.

Erhard notes that a Postcondition failure is an error in the implementation of the task; that is something that should be reported to the task.

Steve suggests that only things that are about the subtype of an actual parameter should be outside. Tucker and Erhard concur.

Steve will look at this to cover Type Invariants and make sure that the exception propagates to both places.

Approve intent of AI: 9-0-2.

AI12-0194-1/01 Language-defined aspects and entry bodies

Steve drives us to wonder about whether this rule prevents pragmas (like pragma Inline) on renames, bodies, etc. Obviously implementation-defined pragmas can do what they want.

Randy wonders if there was a problem; Tucker claims that this wording only applies syntactically to aspect_specifications, because of the use “on”. Sounds like splitting hairs; but in the absence of a better idea, we'll leave the hairs split. (At least most of us still have hair. :-)

Tucker points out that we don't want aspect Inline to work on renames; the reason for using the pragma on a rename is to avoid naming all of them.

Approve AI: 11-0-0.

AI12-0195-1/01 Inheriting body but overriding precondition or postcondition

This rule essentially matches the way a manually generated wrapper would work.

Steve wonders if “only a call” is enough for a function, as a return is also involved. Steve is in fine form, but we think the wording is fine.

Randy wonders about “equivalent”. The two expressions don't have the same types, how could they ever be equivalent? “Equivalent in terms of the non-inherited functions that are called”? “Equivalent in terms of which non-inherited function bodies are executed”?

“Equivalent (with respect to which non-inherited function bodies are executed)”.

Steve notes that 3.4(27/2) says that you convert only one level at a time. Tucker says that his rule is explicitly different; it matches the wrapper model. If the wrapper directly calls the level of the actual body (and it often would), then you'd skip intervening levels.

Approve AI with changes: 6-0-5.

AI12-0196-1/01 Concurrent access to Ada container libraries

Brad is relying on tampering to prevent insert/remove for the full container during concurrent access.

A discussion about calls on different concurrently called operations ensues. Looking at A(3/4), Steve suggests “any language-defined subprogram” be replaced by “any two language-defined subprogram (possibly the same)”. Tucker would like to clarify further, as the objects need to belong to different calls.

Make a new AI to clean up this wording and assign it to Steve. Tucker will review.

Back to this AI. Tucker suggests that we just use the term reentrant rather than concurrently callable.

Approve intent of AI: 10-0-2.

AI12-0197-1/01 Generator functions

Raphael describes the proposal.

Typo “Prime_Numbers” has “end Even”;

These could be implemented with a task in existing Ada (but that would be expensive).

We discuss the Tokenize example. It is not very convincing example. For these to be more convincing, you need a more complex state where the “program counter” can encode it, or a need for task-safety, or a need to have multiple copies.

Steve asks where yield is allowed (just top-level or more nested units). That matters as terms of implementation.

Jean-Pierre thinks that this can be done with Ada tasking; other languages use this model because they don't have real tasks. Ada doesn't need to add these mechanisms, we have better way to do this.

Raphael doesn't believe that a task is a good way to write a generator; the result is awkward.

Florian says that he uses them often in Python. He says that they are a convenient, light way to do things that would be harder (but possible) to create in existing Ada facilities.

Jean-Pierre would like to see convincing examples. The tokenizer example doesn't need enough state to be convincing. (Randy notes that he just wrote a tokenizer for the ACATS grading tools; the only state is the next character to read, otherwise the initial state for any token is identical – tokens by their nature are context-independent.)

Erhard asks whether 'Next is a protected operation? Does it allow multiple tasks to call it? Probably not, which means that it is not task-safe for a global generator. Raphael notes that it is easy to be use one of these in a task-safe way (because a generator doesn't have to have any global state). But it has to be written that way; there's nothing inherent in the concept that makes it task-safe. (And one would have to protect the generator object itself if one wanted to use a single generator state for multiple tasks.)

Erhard, how do you terminate a generator? (From some external operation.)

The stacks are separate, and they'll live until the object is gone. Global generators would hang around forever. Raphael says there could be a terminate command to get rid of the stack.

How does the routine find out about termination? Some sort of finalization. That's missing in here.

Jean-Pierre says this looks sort like a task. Ed says it is more like a protected object with its own stack.

If the generator object is terminated, the generator is “left”, and that causes finalization. We don't want to talk about exceptions being raised.

Steve would like syntax that works with **type**, much more like protected objects. Tucker disagrees, he wants it to be light-weight.

Ed suggests that a protected type with a single function could play this role. Steve notes that such a function doesn't have a program counter (PC). Ed says that it can encode the state. Tucker says that the value of a generator is that (some) of the state is encoded in the PC.

Jean-Pierre says that this can be written as a task. He views the issue as scheduling, we don't want real threads here. So perhaps we should revisit passive tasks.

Alan notes that scheduling could allow the generator to make progress to the next yield when there is no other work to do. (Lazy is not the same as coroutines.)

Florian notes that a nested generator could use outer local state, and that would be an issue with a task solution (it would not be thread-safe). Generators don't introduce non-dependency. (But they also don't allow parallelization naturally.)

Steve Michell notes that today one wants constructs that can be parallelized.

Ed notes that a generator is lazy evaluation; one could imagine an eager generator which works ahead.

Jean-Pierre notes that we want to be able to do things in parallel if we can do it that way.

Randy notes that Yield is the name of a procedure in Annex D, so making it reserved is an incompatibility.

Randy wonders if that Ada customers actually care about this feature. It's a lot of work, is there sufficient demand for this use?? Erhard notes that he has grad students that won't use Ada because of the lack these sorts of features. Florian notes that they would like these sorts of things for specifications at Altran.

AdaCore doesn't have clients requesting generators in Ada, but of course that they don't usually come up with language features.

Florian finds some examples in his Python code. Raphael notes of a GPS highlighter routine where the rules are very complex. We ask Florian and Raphael to provide some better examples for the use of generators to the group.

Steve would like to propose a different syntax. Jean-Pierre would like to propose something based on passive tasks. Alan notes that we want a more general concept that can express generators (either co-routines or passive tasks), not something specialized.

Keep alive: 11-0-0.

AI12-0197-2/02 Passive Tasks

On Monday, we look at the (second version of the) proposal that Jean-Pierre sent overnight. [This AI number was assigned after the meeting - Editor.]

Alan suggests that there should be only one entry, and it should not allow blocking.

Can the aspect be ignored? No, you would have to guarantee that only one passive task is running at a time, that's not equivalent to ignoring the aspect. You could implement this with multiple tasks, but only one can run at a time.

This is trying to provide a stack and state without having a thread of control. The calling tasks would execute as a proxy up to the point of the interaction (rendezvous). The same would be true for initiation.

Ed wonders that since we defined protected objects to replace passive tasks, couldn't we do this with protected objects instead? But they don't have a state, as they don't have any statements. Someone suggests that we could add state to them; that idea was generally disliked. [I'd still like to see how it would work, it might be more Ada-like than the proposal in AI12-0197-1 – Editor.]

Tucker notes that this is likely to be a task with multiple accepts, so it is encoding the state in the program counter.

Alan thinks needs to be limited to one entry; otherwise the caller would have to block. Which is definitely not a coroutine. Alan thinks that multiple callers would work OK.

Steve notes that you can't get indefinite results this way, as there are no entry functions. Randy notes that we could fix that (add entry functions). Jean-Pierre notes that people have been asking for that for years.

This is an alternative to generators. [The assigned number reflects that – Editor.]

Tucker dislikes this approach, based on his experiences with passive tasks in Ada 9x. Protected objects were invented to provide a clear solution. He thinks the same idea appears here. Florian agrees.

Raphael disagrees; he thinks that a passive task would map cleanly to the examples. Besides, this seems to be a different problem – the Ada 9x passive task was trying to encode mutual exclusion where the execution state just got in the way. In this case, the execution state is central to the problem.

A passive task would probably have to lock, else it wouldn't act as a task. Florian does not want generators to do locking. We diverge into a discussion about protected objects and generators getting mixed.

Florian says that static analysis would be easier for a separate generator construct than for a passive task. The passive task would have possibilities that wouldn't be allowed for a generator.

Alan notes that a passive task can be easily changed to an active task, thus making it possible to easily switch to an eager, active model.

Tucker worries that if there are two tasks calling a passive task, then the passive tasks could be running in parallel. Tucker says that a passive task would have to allow multiple clients, whereas the original generator proposal does not allow multiple clients at all. Randy wonders if we really want to add more multitasking hazards (that is, features that don't work with multiple clients automatically) to the language. We have plenty of those already.

Steve thinks that an entry function could have problems with T'Class, you would have incomparable accessibility. Steve wonders if passive tasks essentially require entry functions. Probably.

Jean-Pierre will improve the passive task proposal, with Raphael assisting.

Keep alive: 6-2-4.

Ed opposes as he thinks that doesn't answer the problem. Florian doesn't want to introduce another feature that is almost what is needed but not quite.

AI12-0198-1/01 Potentially unevaluated components of array aggregates

Summary: “belong{s}”.

Approve AI with change: 11-0-0.