Minutes of the 57th ARG Meeting

16-18 June 2017

Vienna, Austria

Attendees: Raphael Amiard, John Barnes (not Sunday, Saturday late), Randy Brukardt, Alan Burns (not Sunday), Jeff Cousins, Brad Moore, Erhard Ploedereder, Jean-Pierre Rosen (not Friday); Florian Schanda, Tucker Taft, Tullio Vardanega.

Observers: Steve Michell (Friday morning only), Pat Rogers (Friday only), Joyce Tokar (Friday morning only), Johann Blieberger (Friday morning only).

Meeting Summary

The meeting convened on Friday, 16 June 2017 at 11:30 hours and adjourned at 13:15 hours on Sunday. 18 June 2017. The meeting was held on Friday in the Palais Eschenbach. On Saturday and Sunday, the meeting was held in a classroom at the Technical University on Operngasse. The meeting covered all of the normal AIs and some of the amendment AIs on the agenda.

AI Summary

The following AIs were approved:

AI12-0224-1/01 Use of Fortran C Interfacing features (6-0-3)
AI12-0231-1/01 Null_Task_Id and Activation_Is_Complete (10-0-0)

The following AIs were approved with editorial changes:

AI12-0201-1/01 Missing operations of static string types (9-0-0)
AI12-0206-1/01 Nonoverridable should allow arbitrary kinds of aspects (8-0-2)
AI12-0207-1/02 Convention of anonymous access-to-subprogram types (9-0-1)
AI12-0216-1/01 6.4.1(6.16-17/3) goes too far (8-0-3)
AI12-0217-1/01 6.1.1(27/3) should be less restrictive (10-0-0)
AI12-0219-1/01 Clarify C interfacing advice (8-0-1)
AI12-0222-1/01 Representation aspects and private types (8-0-2)
AI12-0225-1/02 Prefix of Obj'Image (9-0-0)
AI12-0227-1/01 Evaluation of nonstatic universal expressions when no operators are involved (9-0-0)
AI12-0228-1/01 Properties of qualified expressions used as names (10-0-0)

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

AI12-0064-2/10 Nonblocking subprograms (8-1-0)
AI12-0079-1/04 Global-in and global-out annotations (10-0-0)
AI12-0111-1/05 Tampering considered too expensive (9-0-0)
AI12-0119-1/03 Parallel operations (11-0-0)
AI12-0211-1/02 Interface types and inherited nonoverridable aspects (9-0-0)
AI12-0233-1/01 Pre'Class for hidden operations of private types (8-0-2)

The following AIs were discussed and assigned to an editor:

AI12-0075-1/04 Static expression functions
AI12-0189-1/03 Loop-body as anonymous procedure
AI12-0210-1/00 Type Invariants and Generics
AI12-0212-1/01 Container aggregates
AI12-0230-1/01 Deadline Floor Protocol
AI12-0232-1/01 Rules for pure generic bodies

The intention of the following AIs were voted, but then were discussed again later in the meeting (the final results are above):

AI12-0211-1/01 Interface types and inherited nonoverridable aspects (10-0-0)

Detailed Minutes

Welcome

Jeff asks to move AI12-0119 to have high priority. No objection.

Randy notes that we have two new priority levels, as too many AIs were being given Low priority. By having two new levels, we have more flexibility in assigning priorities, hopefully so that we separate AIs more by importance.

Vulnerabilities Annex (WG 23)

Alan had asked for some time to discuss the newly added vulnerabilies.

Deep vs. shallow copies. Yes the vulnerability exists, it can be mitigated using limited types or controlled types. Raphael also suggests using a container if possible. Erhard will take this item.

Templates and generics. Needs update. After reading, we agree that no change is needed. Joyce and Erhard will review the main document to ensure all issues are covered.

Inheritance. The new text is about the fact that an overriding method does not need to call the parent item. Tucker notes that we need to admit that, and then point mainly at the mitigation strategies in the main document. Raphael notes that it can be partially mitigated by the use of class-wide contracts. Can't reliably tell whether something is initialized, though. Tucker will take this item.

Liskov. Tucker notes that mitigation is to use class-wide contracts. Tucker will take this one.

Redispatching. Ada default is to not to redispatch. So Ada has the vulnerability, but it requires an explicit action. Tucker will take this one, Erhard will comment.

Polymorphic variables. Upcasts are bad. Downcasts are bad. Downcasts can fail at runtime, that can be a vulnerability. Tucker suggests that the mitigation is to not do unconditional downcasts (that is, check that it is the right type before converting). For upcasts, type invariants will mitigate. The invariant is checked on "exit" from the upcast, so that the specific invariant will be rechecked. Tucker will have to take this one.

We move on to the regular agenda.

Apologies

Steve Baird, Gary Dismukes, Bob Duff, Van Snyder apologize for not being able to attend.

Previous Meeting Minutes

John has some typos (on a printout of the minutes, of course), nothing exciting. Approve minutes with corrections: unanimous.

Date and Venue of the Next Meeting

Tucker will host the next meeting October 13-15 in the Boston area. WG 9 will take their normal slot in the morning.

Ada-Europe has offered their normal slot for the summer 2018 meeting; which will be held in Lisbon, Portugal. Our usual dates following the meeting would be June 22-24.

Thanks

Thanks to Ada-Europe for the fine accommodations on Friday

Thanks to Tullio Vardanega for opening the doors and other facilities issues on Saturday and Sunday.

Thanks to the chair for running the meeting.

Thanks to the editor for taking the minutes.

Unfinished Action Items

Can't ask Steve about AI12-0127-1, AI12-0016-1, and AI12-0020-1, since he's not here.

Florian did not work on AI12-0017-1.

Tucker did not work on AI12-0191-1. AI12-0197-4 is a coroutine mechanism, Tucker does still want to do that. It wasn't explained enough for others to help, so it stays with Tucker.

Raphael was interested in looking at a library as an alternative to AI12-0197-3. Florian would like to continue to persue the syntax solution. He doesn't like magic in a library. Randy notes that Tucker's coroutines feature probably would be used in a library to make a generator. Florian will take over AI-0197-3 as a lower priority for him.

Tucker thinks that stable containers is enough to support the parallel reads (along with the previous changes from AI12-0196-1). So drop this action item.

Current Action Items

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

Raphael Amiard:

- AI12-0212-1 (report on C++ facilities)

Steve Baird:

- AI12-0016-1
- AI12-0020-1
- AI12-0075-1
- AI12-0127-1 (with help from Florian Schanda)
- AI12-0210-1
- AI12-0212-1 (assist Florian Schanda)

Randy Brukardt:

- AI12-0017-1 (assist Florian Schanda)
- AI12-0064-2 (with help from Tucker Taft)
- AI12-0112-1
- AI12-0232-1 (assist Tucker Taft)

Editorial changes only:

- AI12-0201-1
- AI12-0206-1
- AI12-0207-1
- AI12-0217-1
- AI12-0219-1
- AI12-0222-1
- AI12-0225-1
- AI12-0227-1
- AI12-0228-1

Alan Burns:

- ●AI12-0230-1 (with assistance from Tucker Taft)

Brad Moore:

- ●AI12-0119-1
- ●AI12-0197-4 (assist Tucker Taft)

Florian Schanda:

- ●AI12-0017-1 (with help from Randy Brukardt)
- ●AI12-0127-1 (assist Steve Baird)
- ●AI12-0188-1 (see discussion of AI12-0189-1)
- ●AI12-0197-3 (lower priority than others)
- ●AI12-0212-1 (with help from Steve Baird)
- ●New AI for Iterator filtering (see discussion of AI12-0212-1)

Tucker Taft:

- ●AI12-0064-2 (assist Randy Brukardt)
- ●AI12-0079-1
- ●AI12-0111-1
- ●AI12-0189-1
- ●AI12-0191-1
- ●AI12-0197-4 (with help from Brad Moore)
- ●AI12-0211-1
- ●AI12-0230-1 (assist Alan Burns)
- ●AI12-0232-1 (with help from Randy Brukardt)
- ●AI12-0233-1

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 10 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

AI12-0064-2/10 Nonblocking subprograms

Note: !wording and !proposal seem merged here.

Tucker asks our real-time people if this is important. Alan says that it isn’t usually considered, because blocking is rare in real-time systems, But he thinks from a software-engineering perspective, it should be.

Jeff says “blocking restriction expression” is confusing. Randy says it was an attempt to avoid double negatives in the wording. Tucker suggests that we should try “nonblocking expression” instead. Randy doesn’t see any double negatives from a quick search.

This was the end of the Saturday session; this AI was resumed on Sunday morning.

Randy notes that the wording he has doesn't seem to apply to the generic as a whole. That's clearly wrong.

For generic matching: If Caller'Nonblocking then callee'nonblocking is required.

If these are in a generic, you have to deal with the nonstatic cases. Otherwise, not.

If Caller has nonblocking nonstatic then if called routine has nonblocking True, always OK; if nonblocking False then error if in body, else OK with a recheck in spec of instance), else if nonblocking is nonstatic: (OK in spec with recheck in spec, otherwise [in body], poor man's inference).

If Caller has nonblocking True, then if the called routine has nonblocking True, always OK, else if nonblocking False then error; else if nonblocking nonstatic, then error in body, else OK with recheck in spec of instance).

Should we allow other expression forms other than **and**? They aren't really useful, so perhaps we could simply allow only **and** and **and then**. Then the "inference" rule becomes simpler.

Jean-Pierre suggests that we allow **or** if you might arrange to call only the nonblocking version if nonblocking is required and a blocking routine otherwise. But the blocking routine would be illegal in such a case (if the subprogram's nonblocking is True), as there is no flow analysis. So this couldn't happen.

So, nonblocking-static should just allow **and**, **and then**, and parentheses, and static expressions of course. Drop the rest of it.

Approve intent: 8-1-0. Erhard says he doesn't understand the consequences, so he votes against.

AI12-0075-1/04 Static expression functions

Bob Duff had suggested that an expression should be nonstatic if it raises an exception. This only makes the code legal if the expression is used in a context that is not static.

Tucker says that this seems like going the wrong way for our users.

```
N : constant Integer := 0;
if N /= 0 then
   X := 7/N;  -- Currently illegal.
end if;
```

Erhard notes that an expression function effectively has a different rule:

```
function Safe_Divide (X, N) return Integer is
   (if N = 0 then Integer'Last else X/N);

Safe_Divide (7, 0);
```

This is clearly legal (via the statically unevaluated rules).

Tucker has no problem will this call being legal; he doesn't want to change the existing rule to just make the wording easier. Randy notes that some of us would prefer to change it in any case.

Who wants to change the rule: 1-5-2.

So no general change to the rule.

Give back to Steve Baird for completion (including compatibility research).

AI12-0079-1/04 Global-in and global-out annotations

Tucker discusses the changes. He settled on the syntax using an Ada mode. The mode is not optional. He then (after some effort getting a projector and screen to work) showed us some examples from his Ada-Europe talk.

Florian wonders why package names cover all of the visible objects. You could have conflicting or redundant specifications. Randy notes that some of those would be useful (read everything, write something specific). Tucker suggests that if there is redundancy, we should take the more specific one.

Florian wonders if the publicly visible objects should be left out. Tucker explains that you can't name the objects in a package accessed via limited with. He figured it made the most sense that regular with works the same way.

Tucker suggests that possibly we could add some syntax to specify just the private data. Say something like

```
        Global => in out P3 private
```

Out says that if you read, you must write somewhere. **In out** allows either. Tucker explains that it must contain an explicit write somewhere in the entity or something callable. Erhard says that isn't checkable, you can say that it is passed as an **out** parameter, but we can't guarantee that it is written. Tucker will fix this.

Florian says that in SPARK, items declared in limited withed packages can be referenced by name.

Synchronized requires at all reads/writes are of protected or atomic objects.

Erhard is concerned about task local variables.

```
        G1
        task body T
           G2
           procedure P with Global => in out synchronized is
                 G2 := ....; -- Illegal
```

It is not task safe to access G2 here (there could be a nested task). Erhard groans about this.

Parallel (see AI12-0119-1) is allowed if the non-synchronized global objects are disjoint.

Tucker wonders if **synchronized** should be used as a qualifier, because one wants to tell the reader what is used.

Erhard thinks that **synchronized** ought to go before the mode. Florian agrees, noting that it makes it clearer that it applies to an entire list.

For packages, **private** should also be a modifier.

Tucker notes that we allow record components in these specifications.

Florian notes that Global in SPARK never refers to protected components, even in nested subprograms.

Tucker notes that you don't have to include constants in these contracts.

We talk about non-constant constants a bit – some types are inherently mutable. Tucker notes that "inherently mutable" isn't usable in a Legality Rule, because it isn't determinable (without violating privacy) if a type is controlled. So we can't use that by itself. But we do need to do something with such not-quite-constants.

Tucker had **proof in**, he now wonders if we should leave it out. Perhaps allowing an implementation-defined modifier would be a good way to allow SPARK to do this.

These are "ghost entities" in SPARK. These only appear in the proofs, not in the production code.

Brad wonders if this is where parallel dependencies for loops would be capture the check. The requirements will be in the parallel operations AI. (AI12-0119-1)

Erhard asks for time for off-the-wall comments. He wonders why we have to do this in the language. Can't we synthesize the global information? Florian notes that synthesizing these annotations is very hard and error-prone.

[Editor's post-meeting note: Synthesizing global information necessarily requires having the body available; that completely defeats the purpose of having separate specifications which is a cornerstone of Ada.]

Tucker notes that we could encourage the creation of tools to annotate the source semi-automatically.

Raphael thinks it is outside of our jurisdiction.

Erhard worries that the model would not be maintainable. If the lowest level has to be changed, one has to be changed all of the uses.

Florian notes that abstract state made SPARK scalable. Global has the appropriate properties to abstract things.

Tucker suggests that **synthesized** as an option, which would be optional to be implemented. That would allow implementations to provide it, provide some encouragement, and the default would be that it falls it back to **all**. To be fully portable, code would need explicit annotations.

Synthesized versions would appear mainly in diagnostics; it doesn't change the source code.

Erhard notes that we will not parallelize unless all of this work is accomplished. It seems like too much.

Approve intent: 10-0-0.

AI12-0111-1/05 Tampering considered too expensive

Change the subject to say something about Stable containers.

In the stable view, there is no operations that tamper with cursors. So there are no tampering checks needed when using the stable view.

Having this did save quite a bit of time when Florian experimented with GNAT. (He had to disable some GNAT optimizations to figure this out.)

Randy and Tucker tries to figure out how to handle the different tampering with elements changes. They have a bunch of different ideas, not clear which is best. They eventually converge on listing the operations in the indefinite containers that prohibit tampering with elements (along with a definition of that term and the [additional] operations that make checks); the definite containers will only talk about tampering with cursors.

Tucker asks whether we need to given the entire package syntax, or just refer to the original operations. Randy notes that we give the entire package definition for the vast majority of language-defined packages, even though the text often could be determined from the rest of the rules. In other cases, we don't give any part of the package definition; there's never a partial definition, so that would be new if we did it here. He thinks we should give the entire definition to be consistent.

Erhard agrees with Randy that the full package syntax should be given.

Tucker wonders if we should have a separate clause, or just mixed in. Randy notes that we don't have any 4 part clause names. So we should mix it in.

In A.18, talk about tampering in general, and stable packages in general. Then put the rest of the text in with the rest.

Approve intent of AI: 9-0-0.

AI12-0119-1/03 Parallel operations

Brad tells us about the changes: Parallel blocks hasn't changed at all. Parallel loops is simplified to just "**parallel**" - no reductions. Parallel reduction expressions are introduced.

Tucker notes that there is no syntax for parallel container iterators. Brad says that was intended, it's just missing. The syntax needs to have **parallel** completely optional.

loop_parameter_specification ::=
  defining_identifier **in** [**reverse**] {[**parallel**]} discrete_subtype_definition

Florian wonders about complicated reductions, there doesn't seem to be a way to do that. He starts explaining a map-reduce, discussing chunks and threads. Tucker notes that you want to think sequentially (don't think about threads); the compiler does the chunks/threads.

```
(for each Element of Set => <""> operator F(Element))
```

The operator has the special aspect (to allow things in parallel).

Tucker notes that without **parallel**, then the computation for a parallel reduction expression is sequential (although the compiler might add some parallelism anyway if it can prove that it is safe). Giving **parallel** tells the compiler that the computation can be parallelized, but that is not mandated.

The operator is not required to be commutative, so the results have to be presented in the appropriate order. The operator needs to be associative to allow combining in parallel.

We have to decide where **parallel** appears; it should be in the same place in all.

```
for parallel I in 1..10 loop
for I in parallel 1..10 loop
for I in 1..10 parallel loop
parallel for I in 1..10 loop

for I in reverse parallel 1..10 loop
```
-- *But this is dubious, as the order shouldn't matter.*

So perhaps reverse and parallel should be optional, but not allowed together.

```
1) for parallel E of A loop
2) for E of A parallel loop
3) for E of parallel A loop
4) parallel for E of A loop
```

If you exit from the loop, other chunks are terminated. Tucker notes that this was discussed at IRTAW and the conclusion was that it had to be that way.

Straw vote: like-hate-neutral

1) 8-1-2
2) 1-5-5
3) 2-5-4
4) 5-1-5

Jean-Pierre reads the first as "**for parallel** I".

Florian notes that 4 is similar to a parallel block. Tucker notes that actually it is ambiguous with a parallel block. Specifically:

```
parallel
   for ...
```

This could be a parallel block with a for loop in it, or a parallel loop.

There is a suggestion that we make it **parallel begin**, which would eliminate the ambiguity. Then every parallel item would start with **parallel**.

Brad hates 4. He felt that the main construct is **for**; **parallel** is just an optimization. You don't want to start with an optimization.

Preference vote: 1-4-neutral: 2-7-2

Moving on to the angle bracket syntax. That represents the initial value.

Randy thinks that it would be hard to read and write (even if it parses). Tucker doesn't believe this. He thinks it works just like any unary operator. Randy is skeptical.

Tucker had suggested using @ for this (assuming the initializer comes from somewhere else), but that doesn't work well if this was used in the RHS of an assignment.

Tucker doesn't believe that there would be any practical conflict with relational operators. It doesn't make a lot of sense to use ">" as a reducer.

Associativity is necessary, in order to use a tree reduction. Commutative is not. The classic example of a non-commutative operator is “&”.

If the reducer is not a symmetric binary operator, then it can’t be done in parallel. (But it still would be supported.)

Florian would like to have an aspect that declares it as commutative. He notes that such an aspect should work on generic formal.

Should the parallel block have a declare part? Something like:

```
declare
parallel begin
end;
```

This is semantically different than a usual block. [We don't seem to have decided this question at this time – Editor.]

Someone suggests using **par** rather than **parallel**. That’s shorter, which makes it more likely to conflict with existing code. With the exception of **elsif**, Ada uses full words for reserved words, never abbreviations. This would be a strange place to break that policy.

Erhard suggests using {} rather than <>. He notes that {} means repetitive usage. <> is a placeholder, and arguably <init> is a placeholder.

Florian suggests having the initial value be an aspect. But there is still an empty space in the syntax, something has to go there. Florian claims that nothing is OK. Yuck.

There has to be a single <> in the reduction expression. (case <> is when 1=> ...)

Randy wonders about confusion with quantified expressions. A quantified expression has “some” or “all”. And logically, it is a special kind of reduction (predefined), so one doesn’t necessarily need to differentiate.

Returning to the syntax:

Florian suggests <>, with an identity aspect on the function itself.

Jean-Pierre suggests <> => 0 + ... Erhard says that it is highly ambiguous.

Tucker’s proposal doesn’t require the initial value is the identity. Florian believes that it needs to be the identity and needs to be enforced. Tucker actually is asking for both an initial value and an identity for the reduction operation. That’s necessary for parallel operation of the reduction (each chunk will need a starting identity).

We discuss other languages solutions to this. Several others use lambdas. Erhard comments that using lambdas requires a lot more text.

Florian wonders if we should allow just using <> to represent the identity value, and it allows an initializer as an optimization. That seems like an additional option.

We discuss writing a parallel matrix multiply. How do you write the identity?

Straw poll: The item is: (1) initial value; (2) identity; (3) initial value with shorthand that means identity.

Preference: (1) 2, (2) 0, (3) 9

Syntax options for the item in the reduction expression:

```
(1)  <<   >> <<>>
(2)  <   >    <>
(3)  [   ]    []
(4)  {   }    {}
```

Erhard worries about:

```
< a > b >
```

This could be solved by making the inner part simple_expression. Then a relational operator would have be parenthesized. `<(a > b)>`

Straw poll: Love-Hate-Neutral on the options given above

(1) 7-2-2
(2) 6-2-3
(3) 0-8-3
(4) 4-6-1

Drop 3 and 4.

Prefer (1) 5, (2) 6.

We continue to discuss without any choice.

Parallel blocks:

```
[block_id:]
[declare declarative_part]
parallel begin
   sequence_of_statements
and
   sequence_of_statements
and
   sequence_of_statements
parallel end [block_id];
```

We now start arguing about whether the **parallel** belongs before the end. It seems unnecessary. Randy comments that we don’t have **parallel** before the end of a parallel loop.

[Editor's post-meeting note: In comparing this to the regular block, the regular block uses handled_sequence_of_statements. It seems that there should be a handler somewhere (probably before the end).]

Tucker argues that it is necessary here because of the **and**s. The loop is not special. Erhard agrees saying that that is necessary in order to close the grouping.

Florian argues that **end parallel** reads better. The example in the AI uses that.

Straw poll:

parallel end: 5, **end parallel**: 3, abstain: 3

What about loops?

```
end loop [id]
end parallel loop [id]
parallel end loop [id]
```

Jean-Pierre notes that we don’t try to distinguish between the kind of loop (for, while, unconditional). Several people don’t want unnecessary keywords. Florian notes that people want to be able to just add “parallel” to an existing loop without having to find the end. It is a modifier more than a declaration.

Straw poll for Favorite for the ending of a loop: **end loop** 8, **parallel end loop** 1; **end parallel loop** 2.

Straw pool for favorite for the ending of a block: **parallel end** 4; **end parallel** 1; **end** 6.

Alan wonders if there is any way to control the implementation of these things. Not currently. Tucker says that he was proposing to add some “stuff” after the keyword **parallel**, for that sort of control. But that should be a separate AI.

Erhard wonders what the intent of handling data dependencies. Tucker says that the default is that it checked that there aren’t data dependencies. Essentially, the use of **parallel** is an assertion that the loop or block or expression *can* be parallelized.

Approve intent of AI: 11-0-0.

On Sunday, Brad asks to discuss this again. He had a new idea, sent in e-mail on Saturday night. He essentially eliminated the separate Reducer and Identity aspects by requiring that the top-level expression has a specific form.

Tucker doesn't think this helps anything. Brad thinks it is simplifies the proposal.

Tucker thinks that it makes this harder to use the construct. Jean-Pierre thinks that it works better if there is the possibility of no operation (that is, a zero-trip loop).

Raphael notes that the example at the end of Brad's e-mail creates extra unbounded_string for each iteration. That is hardly going to help performance. Randy and Florian both comment that the second version seems complex, no one would write that if they didn't have to.

Florian also comments that people used to Python are used to a general form similar to the original proposal.

So we will not use this idea.

AI12-0189-1/03 loop-body as anonymous procedure

Should be difficulty Hard.

Tucker re-explained the idea. Jean-Pierre wonders where the types come from; from the process function. You can repeat them (but don't have to).

Raphael thinks this is bad idea. He goes over the advantages: (1) Ability to use existing library functions; (2) easier way to write iterators; (3) the map problem. He makes a pretty good case for this feature!

But he thinks this is way too complex for the benefit. He would rather come up with a generalized iterators that would allow iterating pairs directly. Then there would be much less mechanism.

Tucker does not think that the map problem is the main reason for supporting it. It's useful to have a state that you don't have to try to put into an object.

Raphael would rather add iterating aspects similar to those that GNAT has internally.

Tucker notes this kind of iterator has been written for a long time. There is a lot of code out there, and it would be a much simpler to read and write than the explicit code.

Tucker believes that exceptions/ATC mechanisms would be sufficient to implement exit, goto and the like.

Florian believes that the code is easier to read and understand with this construct. Raphael disagrees. He thinks a version using a lambda would be better. (Yes, this is the third different replacement that he's suggested would be better.)

Raphael shows his lambda version:

```
MyMap.Iterate (procedure (Key, Value) begin
      Put_Line (My_Key'Image & "=>" & Val'Image);
   end);
```

Tucker believes that allowing statements inside of expressions makes things very challenging, both in readability and in syntax. (Syntax correction becomes a nightmare.) The loop body works better because it is a natural place to put statements.

Florian would like to work on AI12-0188-1, to see what issues can be solved with that sort of proposal. He would want to consider something like the GNAT aspects.

Keep alive: 9-0-0

AI12-0201-1/01 Missing operations of static string types

Tucker would like to eliminate the length limit on static strings. There doesn't seem any need for it these days. Moreover, it only applies to declarations, which is bizarre, so that a bunch of concatenations are static no matter how long the result. So it really serves no purpose (capacity limits are allowed by 1.1.3).

Thus, part of 4.9(24) should be dropped.

Brad has typo "we should {be} extending". Also, fix "euqivalent" in the discussion section.

Approve AI with changes: 9-0-0.

AI12-0206-1/01 Nonoverridable should allow arbitrary kinds of aspects

"convinient" should be "convenient".

John would like an example. Tucker says that there is no syntax involved, so that doesn't make much sense.

Drop the editor's notes.

Approve AI with changes: 8-0-2.

AI12-0207-1/02 Convention of anonymous access-to-subprogram types

Need extra curly brackets in:

> The calling convention for an anonymous access-to-subprogram parameter or anonymous access-to-subprogram result is *protected* if the reserved word /protected/ appears in its definition; otherwise, it is the convention of the {entity}[subprogram] that has the parameter {or result, unless that entity has convention *protected*, *entry*, or Intrinsic, in which case the convention is Ada}.

The * and / are backwards in this paragraph.

John would like to generalize the subject: "Convention of anonymous access types"

Brad: there are a couple of typos in the discussion. 2nd paragraph "thaat". 5th paragraph, "...we want [to] conventions..." Also, "...say that convention{s}..."

B.1(22): "which" should be "that".

Approve AI with changes: 9-0-1.

AI12-0210-1/00 Type Invariants and Generics

Tucker and Florian discuss what SPARK would do here.

Give this to Steve Baird.

Keep alive: 9-0-1.

AI12-0211-1/01 Interface types and inherited nonoverridable aspects

The !summary is missing.

Tucker wonders if "match" is defined in generic formal packages. It is, but specifically for various uses. We then wonder if full conformance would work, but no, that does not allow different expressions that are the same value.

He then suggests adding "statically matching expressions" in 4.9.1, and then use that in formal packages and here. Tucker will take a shot at that (perhaps during this meeting).

Approve intent of AI: 10-0-0.

AI12-0211-1/02 Interface types and inherited nonoverridable aspects

Tucker sent a new draft on Sunday morning.

13.1.1(18.2/3):

Erhard wonders if the “denotes the same declaration” is an implication or an additional requirement. Tucker claims this an implication.

Anyway, he’s defining “confirming” for aspects that are names.

Erhard worries that this (as written) doesn’t allow a different form of the name.

```
type T ... with Variable_Indexing => F;

type T2 is new T with Variable_Indexing => P.F;
```

Tucker would like to restrict these to direct names. Erhard and Randy note that if you have it in the general case, it has to work in for any aspect that you might be able to invent. The old wording only was used for nonoverridable aspects, so it could assume how inheritance works.

That suggests that the wording is better where it originally was.

Tucker will take this back, and try again leaving it where it was.

He then asks about the wording for the new paragraph after 13.1.1(18.4/4). This is rather clunky, it may be better without using confirming. A new term (like match was) might be better.

Approve intent of AI: 9-0-0.

AI12-0212-1/01 Container aggregates

Ada doesn’t have an easy way to create a container object if there are more than one element.

Raphael would like someone to look at the C++ facilities for this. He will take an action item to do that.

Tucker would like to have separate aspects for empty and add.

Tucker would like to allow () for an empty aggregate.

Randy notes that we probably want the full generality of array aggregates here. Both for maps and vectors. For instance, (1 | 2 | 7 => bar) should be allowed.

Tucker agrees with Randy that it should get as close as possible to array aggregate syntax, with positional and named possibilities. Probably should have separate aspects for positional and named additions (enabling either or both for a particular container type).

Filtering of index values:

```
for Item of X when Item > 1 => Item-1
for I in 1 .. Num when I mod 2 = 1 => I*2
```

This seems to be a generally useful capability that isn't related to container aggregates. It should be added to all iterators if we do it at all. So, filtering should be a separate AI. Florian will take an action item to create such an AI.

Brad notes that these look a lot like reduction expressions: Tucker says that one could imagine an aggregate constructed by a reduction expression:

(**for** E **of** C => Include (<empty>, E));

Florian also shows a nested iterator:

```
W : My_Set := (for A of X =>
                   for B of X => A * B);
```

Tucker thinks that his essentially would be the same as a nested aggregate. Florian doesn't believe that. Tucker thinks that

```
W : My_Set := (for A of X => (
                          for B of X => A * B));
```

is the same. Florian replies that a set of sets isn't the same as a set.

Erhard suggests putting a comma rather than the arrow, the arrow means that there is a component. Thus:

```
W : My_Set := (for A of X, for B of X => A * B);
```

Maybe the second **for** could be dropped. Florian thinks the inner **for** is new. Perhaps no separator is needed. Or semicolon – but that is too final.

A comma doesn't work because it means sequencing for an aggregate.

No separator seems to work:

```
W : My_Set := (for A of X for B of X => A * B);
```

Tucker wonders whether parentheses would be better:

```
W : My_Set := (for (A of X; B of X) => A * B);
```

The connector could be , or "then".

Florian is concerned about adding filtering to this. Tucker suggests that it goes at the end:

```
W : My_Set := (for (A of X; B of X) when A /= B => A * B);
```

Florian thinks that as the second is subsidiary to the first, and thus the second may not want to execute at all for some first (the set may not exist). So the **when**s should be inside.

Tucker thinks we should leave this to Florian (and the separate filtering AI).

```
W : My_Set := (for (A of X; B of Element(A)) when A /= B => A * B);
```

Tucker is thinking we should allow the iterator to have an optional **when** everywhere possible.

Keep alive: 9-0-0.

AI12-0216-1/01 6.4.1(6.16-17/3) goes too far

Randy and Tucker explain the problem. While we usually call these "anti-aliasing rules", aliasing is allowed so long as the effect is the same in any possible call. These rules are mainly intended to eliminate non-determinism in expressions (not anti-aliasing per-se).

In example Make_Ref3, two parameter have the same name. That needs to be fixed.

Erhard wonders why pass-by-copy types aren't the rule here. That is implementation-defined for records/array, we didn't want the rule to be implementation-defined or to have false positives.

Brad wonders if atomic objects, which are required to be passed by copy by C.6(19), should be included. Tucker says that it seems like a corner case and these rules were never expected to be perfect. He also notes that atomic objects are changing frequently and asynchronously, so the order of copy-back hardly seems significant.

Approve AI with changes: 8-0-3.

AI12-0217-1/01 6.1.1(27/3) should be less restrictive

... have values {that} are within the range …

Tucker finds the wording rather unbalanced, but after several attempts, no better wording seems available.

Approve AI with change: 10-0-0.

AI12-0219-1/01 Clarify C interfacing advice

Randy thinks that the “recommended” makes no sense in wording defining “correspondences”. Jeff thinks his suggested rewording is worse. Tucker thinks it is redundant, he’d rather do something shorter.

He suggests combining everything. So modify the original B.3(69/2) as follows:

> An Ada parameter of a record type T, [of any mode,] other than an **in** parameter of a type of convention C_Pass_By_Copy, is passed as a t* argument to a C function{, with the const modifier if the Ada mode is **in**}, where t is the C struct corresponding to the Ada type T.

For B.3(70):

> An Ada parameter of an array type with component type T[, of any mode,] is passed as a t* argument to a C function{, with the const modifier if the Ada mode is **in**}, where t is the C type corresponding to the Ada type T.

B.3(38.1/3) has an extra word:

> ... calling an imported subprogram that [does] is not pure from a pure package causes erroneous execution.

Drop the entire editor’s note.

Approve AI with changes: 8-0-1.

AI12-0222-1/01 Representation aspects and private types

Tucker suggests that the RM wording ought to be indented or otherwise set off from the AARM note modification. Tucker suggests starting with “Adjust AARM notes as follows ...”.

Add “-- Illegal” and “-- Should be illegal” to the examples in the question.

Approve AI with changes: 8-0-2.

AI12-0224-1/01 Use of Fortran C Interfacing features

Approve AI: 6-0-3.

AI12-0225-1/02 Prefix of Obj’Image

The underscores are missing in *universal_real* and *universal_fixed* in the wording.

Approve AI with changes: 9-0-0.

AI12-0227-1/01 Evaluation of nonstatic universal expressions when no operators are involved

4.4(10) “... a {numeric} universal type...” This doesn’t apply to universal access.

3rd paragraph of question, “Since a generic formal type is never static, this comparison operator {might be}[is] evaluated at runtime”.

In the penultimate paragraph of the discussion: "a{n} attribute".

John: 4th paragraph of discussion has “the the”.

Brad: “...would still have to {have} even more range.”

Erhard: AARM Ramification: The only effect of this rule is to allow Constraint_Error to be raised if {the} value is outside of the base range of root_integer or root_real.

We discuss the use of “value” in this note; Randy says that he was echoing the wording in the static expression rules – specifically 4.9(35/2).

Erhard suggests that "the value" become "its value". No objection. But then the first sentence needs to be singular. So:

> AARM Ramification: This has no effect for {a} static expression[s]; {its}[the] value may be arbitrarily small or large since no specific type is expected for any expression for which this rule specifies one of the root types. The only effect of this rule is to allow Constraint_Error to be raised if {the} value is outside of the base range of *root_integer* or *root_real.*

Approve AI with editorial changes: 9-0-0.

AI12-0228-1/01 Properties of qualified expressions used as names

Remove the square brackets (or mark the text as redundant) in 3.10(9/3). That was something that should have been deleted when it was cut-and-pasted.

Change the inserted text for aliased to:

> {A qualified expression is an aliased view when the operand denotes an aliased view.}

In the first paragraph of the discussion, change the second sentence to:

> We want qualified expressions to change the properties of an object as little as possible.

In the penultimate paragraph of the discussion:

> "...is not something [is] that is..."

Fix the cases of the record declarations in the discussion.

Approve AI with changes: 10-0-0.

AI12-0230-1/01 Deadline Floor Protocol

Alan tries to explain the model of DFP. Various tasks are executing with various deadlines. A PO has a deadline floor, which has to be no longer than the relative deadline of any task that calls it. While in the PO, the nearest deadline of the current deadline and the deadline floor of the PO is used. Thus, if a task calling a PO is close to its deadline, then the deadline is not adjusted when entering the PO.

Randy wonders about having a different locking policy, and how that interacts when multiple scheduling policies are used together. Alan says that a protected object in a mixed system would have to have both a ceiling priority and a deadline floor.

This is intended to be complete replacement for EDF. There doesn't seem to be any existing usage of the old model, so there is no need to keep it.

Alan says that the open issues need to be figured out by him, not by us (the ARG). All he wanted to know whether a complete replacement is acceptable. (It is; there aren't any implementations of it.)

Tucker volunteers to help Alan with the wording for this AI.

Keep alive: 9-0-1.

AI12-0231-1/01 Null_Task_Id and Activation_Is_Complete

Approve AI: 10-0-0.

AI12-0232-1/01 Rules for pure generic bodies

Tucker explains the problem and possible solutions. Randy notes that Pure generic specs need to be handled as well, Tucker agrees.

Tucker will take the AI and work with Randy to get acceptable wording.

Keep alive: 10-0-0.

AI12-0233-1/01 Pre'Class for hidden operations of private types

Tucker tries to explain the question.

Randy wonders if this isn't a privacy issue rather than something about untagged private types. Cases involving tagged private types seem to have a similar problem.

Tucker suggests that for this (a NEW subprogram), the Pre'Class should default to True. That is, if there isn't any visible ancestors at the point of declaration. Tucker notes that you can always override a Pre'Class with a weaker one, so this can't be a logical problem.

Tucker will take this AI.

Approve intent of AI: 8-0-2.