Minutes of ARG Meeting 63M

17, 19 February 2026

Attendees:

Tuesday Session:

Voting Members:

Steve Baird, Randy Brukardt, Jeff Cousins [arrives at 10:56 EST], Gary Dismukes, Brad Moore, Jean-Pierre Rosen, Tucker Taft, Tullio Vardanega, Richard Wai [arrives at 11:16 EST].

Non-voting Members:

Ed Fish, Niklas Holsti, Alejandro Mosteo.

Observers:

Ethan Luis McDonough [arrives 11:25 EST]

Thursday Session:

Voting Members:

Steve Baird, Randy Brukardt, Jeff Cousins, Gary Dismukes, Justin Squirek, Tucker Taft, Richard Wai [arrives 12:02 EST].

Non-voting Members:

John Barnes, Bob Duff, Ed Fish, Niklas Holsti.

Observers:

None.

Meeting Summary

The first session of the meeting convened on Tuesday, 17 February 2026 at 10:34 hours EST and adjourned at 13:32 hours EST. The second session of the meeting convened on Thursday, 19 February 2026 at 10:34 hours EST and adjourned at 13:29 hours EST. The meeting was held using Zoom. The meeting covered all of the regular AIs and amendment AIs on the agenda.

AI Summary

The following AIs were approved:

AI22-0154-1/02 Revised resolution of indexing aspects (7-1-1)
AI22-0156-1/02 Minor presentation issues (7-0-0)

The following AIs were approved with editorial changes:

AI22-0150-1/03 User-defined indexing and access types (6-0-0)
AI22-0157-1/02 Fix definition of known-on-entry (8-0-0)

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

AI22-0076-2/03 No_Dynamic_Accessibility_Checks restrictions (9-0-0)
AI22-0151-1/02 Syntax rules disagree with Text_IO.Get (9-0-0)
AI22-0155-1/03 Fixed lower bound for one-dimensional arrays (4-0-2)

The following AIs were discussed and assigned to an editor:

AI22-0147-1/01 Controlling what is a controlling parameter (8-1-0)
AI22-0148-1/02 Bounded and Unbounded Indefinite Objects (7-0-0)
AI22-0152-1/03 Optional objects (7-0-0)
AI22-0153-1/02 Generalized case statements (5-0-2)
AI22-0158-1/01 Range equivalent to two-component record (6-1-2)

The following AIs were discussed and voted No Action:

AI22-0076-1/03 Restricting Dynamic_Accessibility checks (9-0-0)

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

AI22-0150-1/02 User-defined indexing and access types (9-0-0)

Detailed Minutes

Welcome

Steve welcomes everyone to this meeting.

Attendance

On Tuesday, Randy takes attendance. 7 voters, 3 non-voters, and 0 observers. (Two more voters and an observer showed up later.) On Thursday, Randy again takes attendance, and then had 6 voters, 4 non-voters, and no observers. (Another voter joined later.)

Christoph said he is on holiday and will miss both days. Jeff and Richard will miss part of Tuesday; additionally Richard will be very late on Thursday. Justin says he will miss Tuesday altogether. Bob Duff says he will be missing part or all of Tuesday (it was all). Jean-Pierre says he will miss Thursday. Alejandro says he will miss most or all of Thursday (it also was all). Tullio said (after the fact) that he was on a seriously delayed airplane flight, and thus missed the entire Thursday session. Brad said he had a work conflict and would miss Thursday in a message stuck in a spam filter that no one saw until after the meeting.

Previous Meeting Minutes

There were no comments on the minutes: Approve minutes: 7-0-0.

Date and Venue of the Next Meeting

Our next electronic meeting is proposed for April 29-30th. There is a brief discussion of whether we need both days. We don't know that now on Tuesday morning; we'll schedule both as it is easier to cancel one later.

In any case, Tucker has a conflict with the proposed days, which he says he could move if necessary. He suggests May 6-7th. Tullio can't make either day in early May. A quick look at the next week has Tucker having a conflict on May 14th. We settle on May 12-13th., with the usual time (10:30-13:30 EDT [-5 UTC]) and method (Zoom).

Prototype parallel implementation

Ethan gives a status report during the break. (Your editor did not take notes during this report.)

Introduction to Memory Safety and Ownership

[Editor's note: Tucker gave this presentation before the discussion of AI22-0148-1 and AI22-0152-1 on Thursday. It is presented here as it doesn't belong to any individual AI. Tucker expanded some of these topics to provide additional detail to improve understanding. His slides for this presentation can be found in the active AIs Google drive, a direct link is https://docs.google.com/presentation/d/1bA-msJgsdhMZ3YB2JwoZyw7BcQuyxvDn/edit?slide=id.g3934987ffcf_0_0#slide=id.g3934987ffcf_0_0.]

Tucker gives a presentation on Memory Safety. This is now being required in the US in some places (especially in government projects). Ada did make it onto the list of suggested languages. Rust is of course included. Both languages have "unsafe" portions, so they're not quite complete in that sense.

Garbage collected languages are safe in this sense, but they have not gotten much traction in the embedded world (despite a lot of work in that direction). How they work is just too complex and requires a level of trust that isn't possible in many safety-critical environments.

So languages like Ada and Rust are preferred, where allocation and deallocation occur in expected ways.

The notion of ownership goes back quite a way back (it predates Rust). Tuck's slides have some seminal papers on the topic. One of the oldest articles on object ownership is:

> Dave Clarke. "Object Ownership and Containment". PhD thesis. Australia: University of New South Wales, July 2001. https://pdfs.semanticscholar.org/fd6b/662b341df3f3dc4fbfc285c4a02cb8b0919b.pdf

There is pointer ownership (which is what Rust uses), and there is object ownership (which is based on a containment relationship).

Pointer ownership is based on a set of rules that ensure there is never more than one pointer to the same designated object. This is done by using "move" semantics when a pointer is assigned, where the RHS object effectively becomes null when its value is copied into a new object. Short-lived references are permitted, using a "borrow checker" which makes certain that while a short-lived reference exists, the original pointer is effectively frozen and unusable, until the short-lived reference goes away. Owning pointers are called "unique pointers" in some languages, such as C++, where they might be just one of various kinds of pointers.

Pointer ownership provides much of the flexibility of building up data structures using pointers, in that trees can grow and shrink dynamically, with null pointers being replaced by a non-null pointer to some designated object. But it is intentionally limited to the construction of trees and tree-like structures; cyclic data structures cannot be created with owning pointers.

Object-based ownership foregoes the use of pointers completely, at least at the user level (pointers might well be used to implement object-based ownership behind the scenes). With object-based ownership, the model is strictly objects with components and subcomponents. However, to match the flexibility of pointer-based ownership, components must be allowed to grow and shrink, including shrinking down to effectively nothing. This breaks a rather fundamental assumption of some Ada compilers, where components are fixed in size upon creation, though with discriminated records it is possible to replace a composite object with a whole new object with differently-sized components, and possibly of a different variant. In most Ada compilers this involves pre-allocating a fixed size area known to be large enough to hold any variant and any possible discriminant-dependent components. Allowing components to change in size relatively arbitrarily, such as declaring a component to be of subtype String and then being allowed to assign any value of type String to the component, necessarily implies some amount of automated memory management, and the possibility of an overall object no longer having all of its components contained within a single contiguous block of memory. Instead, such objects are presumed to have an associated storage area (storage pool, storage *arena*), and all of the (sub)components are allocated out of that area, in such a way that the object and all of its subcomponents can be found and freed when the object goes out of scope, or is replaced with a completely new value.

In object-based ownership, a component that is allowed to shrink down to nothing, or to start as nothing and then grow into something, is called an *optional* component. In fact, a whole object can be optional. For the purposes of discussion of the AIs AI22-0148-1 and AI22-0152-1, optionality is indicated by appending **or null** in a subtype_indication or a parameter specification, analogous to the way null exclusion is indicated by prepending **not null** in a subtype_indication or parameter specification. Objects or components of a null-allowing subtype are by default initialized to null, analogous to objects of an access type.

Fundamentally, any sort of ownership implies automated storage management. In particular, the user would never perform explicit deallocation. With pointer-based ownership, this make sense because the rules prevent there ever being two pointers to the same designated object, meaning that when a pointer is set to null, or to point somewhere else, the object it is pointing at can be immediately freed, without the use of a garbage collector. With object-based ownership, the general notion of containment of components implies that when you free an object, or set it to null, you free all of its subcomponents as well.

Overall, the real benefit of ownership comes when the primary data structuring mechanism uses ownership. This has a liberating effect on the developer, because they can build up essentially arbitrarily complex, memory safe data structures, without having to worry about dangling references, or memory leakage. Because components can grow and shrink, it is possible to define a node type hierarchy that uses variants or type extension, and then create a (heterogeneous) container of such nodes, and use the index into the container as a safer sort of reference to the contained node. This also allows the construction of cyclic structures, albeit using indices into an indexed set of nodes, rather than direct pointers. This would also allow the creation of safer "cursors" into containers, since there need not be any direct pointers to individual nodes which might disappear during updates to the container.

Short-lived (direct) references are also possible with object-based ownership, though in some sense this is no different than what is already available in Ada in terms of by-reference parameter passing. For components that can disappear completely, the same sort of rules will need to apply as currently apply to discriminant-dependent components, where the component should not be reclaimed while it is accessible via a by-reference parameter. There is a separate AI (AI22-0131-2 – Part_Of aspect) which attempts to relax these rules a bit.

Randy notes that Janus/Ada has done much of this from the beginning for discriminant-dependent arrays. They are stored and managed separately from the rest of the object, and are reallocated if the discriminant is changed by an assignment.

Gary asks Randy how Janus/Ada deals with Sequential_IO/Direct_IO. Randy notes that he never did (those packages raise Use_Error if given a non-contiguous type); the plan is to have a "flattening" attribute, similar to a stream attribute but it would be subtype-specific (as array representations depend on the subtypes involved). Tucker notes this area needs to be discussed, as the block size to use for Direct_IO in this case would need to be defined. [Editor's note: This also affects the flattening generic of Storage_IO, one would hope a similar mechanism would be used for all of them.]

Bounds are necessary for embedded users, and can cause fragmentation if one is not careful.

Steve notes that this is not just a case of copying a feature that other languages already have, but it is an attempt to make a step forward in language-design by embracing the advantages of ownership without the complexity of explicit pointer management.

Randy notes that cursors are essentially array indexes, which is similar to how he learned to program in Fortran. It feels to him like we're going back to the beginning of programming. Someone mentions that early Fortran didn't even have records, so you had to use multiple parallel arrays to put together effectively a "record" of data about a single logical entity. Tucker mentions that in Fortran, the arrays are homogeneous, so you can't easily represent a heterogenous tree of nodes, where each node is of a different variant or type extension. [Editor's note: Randy should have noted that while array indexes as pointers don't end up dangling per-se, it can be easy for them to end up pointing at the wrong element if insertions or deletions occur in front of them. This mistake has recently burned him in an almost impossible to find bug in the Janus/Ada optimizer. One can arrange to detect that mistake, but only at the cost of extra memory and checks. This seems like trading one programming problem for another. It does trade erroneous execution for what Ada would call a bounded error, and that probably is much of the appeal.]

Analogous to the suggestion that we could use a Restriction such as Static_Accessibility_Checks (related to AI22-0076-2) to reduce the number of run-time accessibility checks, we could have similar restrictions for null checks, such as a "Static_Null_Checks" Restriction or a "Static_Index_Checks" restriction. What this would mean is that the programmer would have to make certain, perhaps with a case statement (see AI22-0153-1 Generalized Case Statement) the object is null-excluding before using it in a context that doesn't permit null. This would provide the same level of safety as languages that require an optional object to be explicitly "unwrapped" before use.

Gary asks if this is similar to what Tucker did in Parasail. Tucker says it is, but there is no bounding capability in Parasail.

Tucker thinks that the use of object-based ownership will eliminate most user-defined finalization as well as use of explicit Unchecked_Deallocation. He says that's because most finalization is used for memory management. (Of course, the implementation might very well be using finalization to do the memory management in some cases – for instance, a storage pool is controlled and it is likely that its memory would be reclaimed when it is finalized.)

Unfinished Action Items

There is no discussion of these items.

Current Action Items

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

Steve Baird:

- AI22-0076-2

Randy Brukardt:

Editorial changes only:

- AI22-0150-1
- AI22-0157-1

Christoph Grein:

- AI22-0151-1 (Request help from Tucker Taft if needed)

Tucker Taft:

- AI22-0063-1 (Split work with Richard Wai)
- AI22-0071-2
- Revise AI22-0119-1 (see Unfinished Action Items of meeting #63I [May 2025])
- AI22-0131-2
- AI22-0135-1 (AI22-0135-2 is on hold)
- AI22-0147-1
- AI22-0148-1
- AI22-0151-1 (Help Christoph Grein with wording on request)
- AI22-0152-1
- AI22-0153-1
- AI22-0158-1
- Github Issue #139

Richard Wai:

- AI22-0063-1 (Split work with Tucker Taft)

Detailed Review

The minutes cover detailed review of Ada 2022 AIs (AI22s). The AI22s 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 202y AARM, the number refers to the text in draft 5 of the Ada 202y AARM. Paragraph numbers in other drafts may vary. Other paragraph numbers come from the final Ada 2022 AARM; again the paragraph numbers in the many drafts may vary.

Detailed Review of Ada 2022 AIs

AI22-0076-1/03 Restricting Dynamic_Accessibility checks

We're not going to do anything with this version, so let's remove it from the agenda.

No action: 9-0-0.

AI22-0076-2/03 No_Dynamic_Accessibility_Checks restrictions

Steve explains the AI. The effect is to change the accessibility levels of some things. The intent is that the dynamic behavior of a program does not change if this restriction is added (if the program remains legal).

Steve was concerned about some wording detail. After a minute, Tuck points out that we need to decide the big picture instead.

Fundamentally, we are "assuming the worst" to eliminate the checks. Steve notes that you have the choice of making the caller or the callee assume the worst, so it is not a clear-cut single possibility for what "worst" means here. That is, one can assume the worst inside of the subprogram, and make no limitations on the caller, or alternatively, one can assume the worst at the call site, and then there are no limitations inside the subprogram.

Gnat has an implementation, but it differs a lot from this proposal. There are some issues with a customer using the Gnat implementation. Steve thinks this proposal is less restrictive (and thus more useful) than the Gnat version. In particular, Gnat treats access discriminants as having the level of the type, which is less useful than the proposal.

Tucker has suggested that this be treated as additional static checks (and not be partition-wide). He suggests "Static_Accessiblity_Checks", where we do not specify that you can eliminate the dynamic checks. Rather, the compiler can decide to do so as an optimization (because it can tell that they cannot fail).

Steve thinks that the restriction has to be partition-wide in order to remove the checks. Randy and Tucker think that a global setting of a restriction should be enough versus setting it for individual files. But Steve (and others) note that "Global" settings can be different in different directories that are created into a single program. So just using the position (that is, in or not in GNAT.ADC file) to determine the full program applicability is not enough for GNAT.

Steve thinks that we might need two flavors of this, so that we can know if it is partition-wide (as that allows omitting some checks and overhead). Restrictions can have parameters, so we agree that there should be a parameter to the restriction as whether it is partition-wide.

Tucker wonders if we could have prototype implementation of this using GNAT. We need to evaluate this set of rules. Steve and Gary think that might be possible, due to customer interest, but they are not in a position to make any promises.

Steve should update the AI with the non-partition-wide version of the restriction, as outlined above, and check if there are any major problems with those cases (presuming that dynamic checks are still occurring).

Approve intent: 9-0-0.

AI22-0147-1/01 Controlling what is a controlling parameter

Tucker explains that some projects want to avoid having things being controlling parameters or results for a tagged type. One usual example is constructor functions. Another case occurs when there are two tagged types in the same scope and a routine that operates on both: one of the parameters must not be controlling.

The usual fixes of using class-wide types (which potentially introduces runtime dispatching, which is not allowed in some environments) or putting the subprogram into a nested package are clunky at best.

Tucker proposes "Rooted" as an attribute that would work a lot like "Class".

Ed wonders if a "controlling parameter" aspect would be better. Tucker says that is roughly what this is doing.

Steve suggests specifying the controlling parameter of a group of tagged type parameters. Tucker hates that, because it seems reverse to the way we normally do these things. Steve says this is an aspect of the subprogram. Tucker says that we don't usually identify the controlling parameters, they just happen based on the type names.

Steve thinks this is an inappropriate use of an attribute. He would want some other way.

Jean-Pierre suggests using **abs** in the declaration to specify non-controlling. Some people like that. A joke (I hope!) is made about using **xor** to specify non-controlling.

Keep alive: 8-1-0. Randy opposes: Looks like a lot of work to the relatively complex inheritance mechanism, which outweighs the minimal gain. (In most such cases, a nested package suffices, either for the subprogram itself, or for an entire ADT.)

AI22-0148-1/02 Bounded and Unbounded Indefinite Objects

Tucker noted that you have to specify in some way to allow an object to be mutable. We can't do that by default, as that would be a substantial (and silent) change in the effect of code.

He proposes an Extent aspect for this purpose, with the values Unbounded, Bounded, and Constrained.

"Bounded" is similar to a bounded holder. "Unbounded" is similar to an unbounded holder. "Constrained" is constrained by its initial value.

Objects of "Bounded" and "Unbounded" extents have to be initialized.

Ed thinks that there are multiple ways to look at a bound (number of nodes, number of characters, or just bytes). Tucker has plans for attributes to determine the appropriate storage size to use for a bound. (He hasn't written that AI yet.)

Bob says that the default would be unbounded for indefinite components. Why not for objects that are currently illegal? Tucker says they have to be initialized (even components). Otherwise, you do not know how much memory to allocate to the object. (The proposed optional objects could be different, but that's a different AI (AI22-0152-1).)

Randy asks about the checks when passing a constrained object to a subprogram that is expecting an unconstrained object. There are none defined, and that seems necessary to avoid nonsense (and also the point of doing this on parameters, to avoid runtime checks in subprogram bodies by moving them to the call site). Tucker has yet to define this.

Steve notes that Gnat allows mutably tagged objects, and those have similar issues when passing to a less constrained class-wide type. Those checks are relatively complex; it requires more than a simple Boolean check.

Randy worries about checks for nested extensions and similar cases where we say "it's great that components can't be unconstrained/indefinite". There needs to be some attempt to identify such places and deal with them. He thinks that coextension and access discriminant checks, for instance, need to be revisited.

Jeff would like a longer justification for this AI. Motivating examples would be great.

Randy suggests that Tucker's slides be put on the web (perhaps with more explanation), and then that can be linked from each of these AIs. [Editor's note: this was done, see the general presentation at the top of these minutes for information on finding it.]

What to vote? Randy suggests "Keep alive", as he remembers the Niklaus Wirth Turing Award lecture, where he talked about how PL/1's original design was elegant and wonderful. But when it got actually defined in detail, it was full of special cases and other warts. He notes that lecture has always influenced his (Randy's) approach to language design. This looks great at the 10000 foot view, but we need to get a lot closer before we can say that it is going to work out. The devil is in the details. The group agrees that Keep Alive is sufficient.

Keep alive: 7-0-0.

AI22-0150-1/02 User-defined indexing and access types

Randy tries to explain the problem and rules.

Ed wouldn't allow the dereference on the indexing for these cases. That is not desired, it would prevent expected operations in the body of the package.

Tucker suggests just disallowing all private types, as there is concern about the oddity of the rule. Steve notes that it isn't clear how instantiations are handled.

Just disallowing all private types would be easy, but way too strong. That would prevent, for instance, an access to a list or tree container being used here. [Editor's note: Actually, all of the predefined containers other than holders and queues have defined indexing operations, so they are banned by the rule preventing access-to-indexed-container-types. Sorry about providing such a bad example during the meeting. Of course, any other private type, such as File_Type, the window types in Claw, and many others, have no technical reason for being excluded here.]

Tucker will try to come up with a more acceptable rule. And we'll revisit on Thursday.

Approve intent: 9-0-0.

AI22-0150-1/03 User-defined indexing and access types

On Thursday, Tucker explains the wording change. Steve suggests "within the declarative region immediately enclosing the access type declaration" as he thinks that is the usual way to put this sort of thing. We check elsewhere in the RM and agree that is the usual way to put this.

Approve AI with changes: 6-0-0.

AI22-0151-1/02 Syntax rules disagree with Text_IO.Get

This is Christoph's AI, but since he couldn't be here, Randy explains the issue. If the syntax is in error, then reading stops immediately and Data_Error is raised. Otherwise, the whole value is read and then any other checks are made, both for any Legality Rules and for the specific checks demanded by the RM. Portability thus requires determining which rules are syntax rules and which rules are Legality Rules so that the point where reading stops is clearly defined. Christoph had noticed 3 cases where the answer is not obvious.

Issue (1) has been checked in the ACATS since 1986, so we need to match the language to the ACATS and compiler behavior.

The intent of (3) seems similar to (1) as it is about "-". So it should read the number, then raise Data_Error. That agrees with the result of 3 out of 4 compilers.

The intent of (2) seems questionable. We discuss a bit the use of this parameter. Someone says that it is for reading tables of numbers. [We didn't discuss the reason for the behavior that it appears is required by the RM: If one gets the table alignment wrong such that an extra column of spaces is included, the error is detected immediately. If trailing blanks are skipped, then the error will be ignored, and it is very possible that one could get only parts of values from following columns as the alignment gets further and further out of alignment. This happened to me once many years ago and the error prevented proceeding with incorrect values (especially a problem with integer values). – Editor.]

Several people argue that this should be handled the same as the Value attribute, which does skip blanks at the end.

Straw vote: Allow extra trailing blanks: 10
Rejecting trailing blanks: Randy. He says Christoph agreed (note the proposal in the AI) - the group says he should have shown up to make his case and vote here.
Abstain: Gary, Alejandro.

The AI goes back to Christoph. Tucker will help with wording if needed.

Approve intent: 9-0-0.

AI22-0152-1/03 Optional objects

Tucker explains the issue. Objects often need an "unknown" value, especially before they are set to a value.

The idea is that there is a value **null**, which every type has, and is only allowed to be used if the subtype is specified as **or null**.

Steve worries that this is adding a property via a subtype. Randy explains that all types have this value, but first subtypes don't have it. So you are just exposing something that you already have, but can't see by default.

Tucker says that integers have an infinite range logically, and what gets exposed for a specific subtype can vary larger or smaller. This is a similar idea.

Tucker notes that he doesn't want to add **or null** to all type definitions. That's a lot of syntax. Richard notes that he thinks people would be surprised if they couldn't do this on a type. Moreover, we did allow **not null** on a type, even though it almost never is useful on a type. This is much more useful on a type (if you want to use this rather than an Unknown enumeration literal, you probably want to declare that fact on the type). So this specification should be available for types (especially as you can make it available by deriving from a subtype that has **or null** -- being required to use a derived type just for this reason seems like a perverse negative incentive). Niklas suggests using an alternative syntax for this. [His idea, not recorded at the meeting, is described in a post in the Github Issue associated with this AI, #149.]

Generic formals need some sort of matching (especially if they allow **or null**, as we are now suggesting).

Access types already have a **null** value. One can say **or null** on access types. Essentially, you can always say **or null**, even on subtypes that are already **or null**.

Steve wonders what happens with interface types that don't agree about this property. [Editor's note: That doesn't seem to be a real problem, as all types have **or null**. And any subtype can add it. The only issue would be that a formal (sub)type that has **or null** needs an actual (sub)type that also has **or null**.]

The recursive type ban does not apply to optional components. There could be multiple types involved.

Gary wonders whether this should be available for limited types. Tucker thinks that it would be useful for parameters. Steve says that a function result could define a limited data structure via build-in-place.

We don't want implementations to add an indirection for all such objects. A single extra bit is enough for many types (although alignment could very well waste additional space).

Steve notes that a type that cannot have **null** might have to stream differently than a type that has **null**. That would be bringing subtypes into streaming. We also would want to avoid changing streaming for existing types as much as possible (that could break existing code).

Richard thinks this is a form of constraint; Tucker doesn't want to use this term for this. **Not null** is not a constraint. Randy notes that has been a continuing source of pain, as a lot of wording has had to be changed to say "constraint, null exclusion, or predicate". Someone notes that we could look for phrases like "null exclusion" to find all of the places to change.

Randy is concerned about using subtypes in resolution. Doing that is new, and would require a lot of changes. He also thinks the subtype of an expression is not well defined in a number of cases. Introducing ambiguity is annoying, but it may not be common enough to worry about. So it may be OK to use a type-based resolution. Prototyping is necessary to decide this. Randy suggests that a membership could be used instead [which also has the advantage of not requiring any use clause to get visibility for non-local types – Editor], an idea which seems to have excited absolutely no one.

Bob suggests compatibility is more important than implementation and language complexity.

Recursive declarations do imply automatic storage management; in this way this AI depends on AI22-0148-1.

The recursive capability isn't essential (and some sense separate) but it seems like a natural combination to Tucker. Niklas notes that it may not be good for embedded systems; Tucker says we need the bounded stuff for those uses.

The justification should be updated to talk about recursive data structures.

Niklas is interested in solving the small problem (that is, optional objects without any recursive data structures) separately from ownership; he proposes a solution based on implicit variants. Randy says that he isn't interested in the small problem; we've been solving that all along and it is a basic part of software engineering. Ed notes that we want a solution that solves a wide range of problems rather than a bunch of simple solutions that may not fit together well.

The default value is **null** if it is **or null**, even if there is a default value for type. It seems that if someone wants an option for unknown, then that is what they should get by default. A common use for default values is to set the value to a special value defined to mean Unknown or Empty or the like anyway, so in some sense these features are unlikely to be used together.

Steve asks if this would allow a component to have an incomplete type. Tucker has not thought about that. Steve thinks about it briefly and thinks it probably would work (so long as any objects are created when the completion of the type is known, which probably would follow from the incomplete rules anyway). Steve thinks that such a capability would make Taft-amendment types and limited withs far more useful. [Limited withs now have to be used with access types; the entire justification for the anonymous access type boondoggle was to make that easier to use. If we can do the same with optional objects, that will help eliminate the explicit use of access types – Editor.]

Keep alive: 7-0-0.

AI22-0153-1/02 Generalized case statements

Tucker explains the idea, to allow more kinds of case statements.

The advantage is that a case statement is easier to read over a tangle of nested if statements, and that we have completeness and exclusivity of limbs checking.

There is also the idea of labeling a choice (essentially adding a subtype for free).

The proposed short hand for wildcards is discussed. Richard notes that the "shorthand" saves one(!) character. Probably should not have both ways to label the choice, as that primarily will add confusion.

There needs to be examples of the multiple choices capability of this case statement.

Bob asks if you could use a case to differentiate between the various descendants of a class-wide type. Tucker suggests that is a good idea.

Randy notes that a downside here is the need to rewrite large chunks of existing code to get this benefit. The text of a case statement and of if statements is quite dissimilar (the times he's had to change from one to the other – mostly for testing reasons – have been rather unpleasant – even the indentation has to change). He notes that Tucker's paper proves that we could make a completeness/exclusivity of limbs check on an arbitrary set of if statements. That would be more useful on existing code, and could be defined with a pragma or some sort of syntax. (He has plenty of if statements that are intended to be exclusive and complete, a check would be very helpful.)

Randy also notes that he doesn't like aliasing buried in the middle of a complex statement. Ed notes that he prefers using a rename outside of such a case statement, so that he can use the same simple name throughout the statement (and other related statements as well).

Keep Alive: 5-0-2. Jeff, Randy abstain.

AI22-0154-1/02 Revised resolution of indexing aspects

Gary explains the AI. Gnat was allowing cases that should have been rejected by the rule clarified by AI22-0146-1. But customers have code that is depending upon some of those cases.

The wording changes in 8.6 and 13.1.1 are to clarify the rules as implied, and to exclude any formal subprograms that happen to be visible at the point of an aspect definition.

Ed wonders if this should be integrated with resolution; the 8.6 rule *is* the definition of resolution for subprogram names.

Approve AI: 7-1-1. Randy opposed, Brad abstains. Randy says he is only opposing the changes to 4.1.6; he believes that virtually all ADTs need to be in a package of their own (potentially nested), lest many problems occur with inheritance. This problem does not occur with one ADT per package, and the understandability loss is substantial.

AI22-0155-1/03 Fixed lower bound for one-dimensional arrays

AdaCore has implemented a syntax for this. Tucker thinks that is fairly heavy, and can't be used on a parameter.

He is suggesting allowing an aspect anywhere (parameters, object declarations, subtypes, etc.).

He notes that he suggests skipping support for multi-dimensional arrays for this idea. He has a design for it, but it is more complicated, and the underlying problem mostly occurs because of slicing (which is not available for multi-dimensional arrays.

Randy wonders if we allow aspects on parameters. Steve says that we do, we've just never used it (there are no language-defined aspects that work on parameters to date, this would be the first).

Ed asks if we could have a static predicate for this purpose. Steve notes that we need to have the array slide, that wouldn't happen with a predicate. We already allow a dynamic predicate, but that doesn't really work when sliding is needed.

Steve says that qualified expressions would not slide, he was wondering if we ought to have this somehow effect the lower bound of aggregates. Tucker does not want to go there.

Jeff asks when this happens on **in out** parameters. Bob notes that bounds are passed in even for **in out** parameters. Steve notes that subtype conversions would do this sliding.

Randy notes that this is the first time we allow anonymous subtype definitions in parameters. (Actually the second time, as we allow **not null**.) At a minimum, we need conformance checks here. Steve agrees, we require matching on **not null**, and this would be similar. He notes that access-to-subprogram types, formal subprograms, and more would need to be handled. [Editor's late musing: The rules for **not null** conformance, especially in generics, was a lot of work to write tests for and to implement, and especially to test and implement correctly! Getting another similar case is not appealing.]

Steve wonders what happens if you try to constrain one of these guys? Does the low bound have to match?? Randy notes that this is known as compatibility [there are separate "compatibility" rules for each kind of constraint] and satisfies [again, are separate "satisfies" rules for each kind of constraint]. These clearly need such rules.

Steve now wonders if membership does the right thing (it doesn't slide). Should it be consistent with qualification?

Ed asks if there should be a Last. We think that isn't very useful, and Steve calls it orthogonality run amok.

Randy says that he thinks that 4.6 needs some wording to ensure that sliding occurs. Generally, the definition of subtype conversion is in 4.6, and it would be annoying to put some important property somewhere else.

Approve intent: 4-0-2. Abstain Gary, Randy.

AI22-0156-1/02 Minor presentation issues

Randy goes through the AI, discussing all 4 cases.

Approve AI: 7-0-0.

AI22-0157-1/02 Fix definition of known-on-entry

Tucker explains the problems.

Steve notes that generic formal types could have an actual type that has a dynamic predicate that is not known on entry. Use of such a subtype in a membership operation needs to be excluded from known-on-entry.

Tucker adds some wording to handle that case. This turns into a lengthy wordsmithing session. As the editor tries to decide if he needs to call time on the efforts, they finally complete.

Approve AI with changes: 8-0-0.

AI22-0158-1/01 Range equivalent to two-component record

The ".." operation is something that you would like to be able to write, especially in order to have something like a user-defined slice. (The actual slice operation would be just another form of Constant_Indexing, so no other operations are needed to support that.)

Randy suggested in the Github issue that we should treat it as a two component positional aggregate.

So Tucker is suggesting that it is an equivalence with an aggregate. The semantics and error checking come from the equivalent aggregate.

Randy had suggested that there is an aspect that to enable this notation, because it would seem to be an issue to get it on an inappropriate type (consider a position type with components X and Y; you wouldn't anybody to write a position as `F .. L`). Tucker agrees with this.

Ed wonders if we should have a first class range type. Randy wonders what extra that would gain. Ed suggests that it could work in a membership.

Tucker notes that this works in assignments and constant definitions. For instance:

```
C : constant Bounds := 1 .. 10;
V : Bounds;
…
V := 2 .. 5;
```

Ed again talks about a meta-language that would allow the definition of properties without the scaffolding. Tucker thinks that it would be a huge amount of work to define. We agree that he needs to propose how this would work before we can seriously think about it.

Richard wonders about other places where other ranges are allowed (such as iteration and membership). We would need to somehow define the **in** operation for membership and possibly iterators.

Jean-Pierre notes that this is similar to a string literal being equivalent to a character aggregate. So we already have this sort of thing in the language.

There is some discussion about whether there is value to tying the range to a specific type. Tucker is asked if the types of the two values need to be the same. He says no, and compares this to the operands of "+" not needing to be the same type.

Jean-Pierre notes that this would allow a membership test on the result of a function call returning a range.

Keep alive: 6-1-2.

Steve opposes. He doesn’t think that this is anywhere worth the cost in complexity (both of implementation and definition). Gary abstains. He doesn’t think it adds enough value. Jeff abstains. He has similar thoughts.