References: 97-209r1 (rationale), 97-256 (specifications), 98-138 (syntax), 98-229 (draft edits), 98-230 (additional specifications and syntax)

Proposal Note:

This revision of this document includes numerous corrections of minor editorial infelicities in the original, but the principal change is the introductions of the terms "content" and "linkage". A proposal note in the original observed that deficiencies in our terminology made much of 6.3 difficult to write. The subgroup found it also made the section difficult to read. We believe that by introducing terms that distinguish what we need distinguished and treat collectively what we don't wish to distinguish, the readability of this section is significantly enhanced.

The terms themselves have been chosen to be plausible placeholders. They have been used in ways such that, if more appropriate terms are identified, they can easily be replaced, but they are plausible enough terms that they could be left in place if no more appropriate terms come to light.

The Main Edit

104:1-5 Revise section to read as follows: "

6.3 Initialization and Finalization

Initialization is the process by which the status of a data object is established prior to any explicit operations on that object. **Finalization** of a data object is the processing that follows the explicit operations on that object. Every data object in a program undergoes both initialization and finalization. Data objects shall not be referenced, defined, or used in any other way before they are initialized or after they are finalized.

Note:

Although every data object undergoes the conceptual processes of initialization and finalization, for most objects those processes involve no operations and thus have no affect on the object.

6.3.1 How initialization is performed

Initialization of an object consists of two steps, performed in order:

(1) **Preliminary initialization** establishes a default status for the object, based on its declared attributes.

(2) **Overriding initialization** may alter that status, based on explicit initialization declared for the object.

6.3.1.1 Preliminary initialization

The effect of preliminary initialization depends on whether it is the content of the object that is being initialized or the status of its linkage to content. Initialization of object content is performed if one of the following is true:

(1) The object has neither the ALLOCATABLE nor the POINTER attribute.

(2) The object has either the ALLOCATABLE or the POINTER attribute and is being allocated in an ALLOCATE statement.

Initialization of object linkage is performed if the object has either the ALLOCATABLE or the POINTER attribute and is not being allocated in an ALLOCATE statement.

6.3.1.1.1 Preliminary initialization of object content

Preliminary initialization of object content is performed on each element of the entity (treating a scalar entity as a single element), based on the type of the entity:

(1) If the data type is one of the intrinsic types INTEGER, REAL, COMPLEX, or LOGICAL, the element is undefined.

(2) If the data type is the intrinsic type CHARACTER, each character position in the element is undefined.

(3) If the data type is a derived type, the following steps are performed:

(a) If the data type is an extended type, initialization is performed (recursively) for the parent component subobject of the element.

> J3 Note:
>
> It is editorially unclear whether there really is a parent component or only a notation that looks like such a component exists. (This is unresolved issue 19.) Depending on which way this is resolved, this text may need to be changed to say something like the “components from the parent type” instead of the parent component and to allow for their default initialization.

(b) Initialization is performed (recursively) for the explicitly declared component subobjects of the element, in the order those components were declared. Default initialization (4.5) for a component is treated like explicit initialization for the recursive initialization of the component subobject.

> Note:
>
> The preliminary initialization of an object of derived type includes both the preliminary and the overriding initialization of its component subobjects.

6.3.1.1.2 Preliminary initialization of object linkage

If an object has the ALLOCATABLE attribute, preliminary initialization of its linkage affects only its allocation status (6.4.1.2). The allocation status is not currently allocated.

If an object has the POINTER attribute, preliminary initialization of its linkage affects only its pointer association status (14.6.2.1). The pointer association status is undefined.

6.3.1.2 Override initialization

The effect of override initialization also depends on whether initialization of object content or object linkage is being performed (6.3.1.1). Additionally, it depends on which of the following cases is applicable:

(1) There is no explicit initialization (5.1) for the object.

(2) Explicit initialization for the object occurs in the type declaration statement for the object. Such explicit initialization always applies to the entire object.

This case is also considered to apply if the initialization being performed is the recursive initialization of a component subobject, as described in 6.3.1.1, and a default initialization is specified in the component declaration statement for that component. Such an initialization applies to the entire component subobject.

(3) Explicit initialization for the object occurs in at least one DATA statement (5.3.13). Each explicit initialization in a DATA statement applies to a scalar object, possibly a subobject of a named object. There may be subobjects of the object for which no explicit initialization is provided.

6.3.1.2.1 Override initialization of object content

The effect of override initialization of object content further depends on whether the type of the object is one for which an initial procedure (4.5.1.5) has been specified.

6.3.1.2.1.1 Override by assignment

If the type has no initial procedures, override initialization is handled in accordance with the rules of intrinsic assignment.

(1) If there is no explicit initialization, the status of the object remains as established by preliminary initialization.

(2) If the object is explicitly initialized in a type declaration statement or component definition statement, that initialization is assigned to the object, as a whole, in accordance with the rules of intrinsic assignment.

(3) If the object is explicitly initialized in DATA statements, the identified scalar objects (possibly elements or component subobjects of the object) are assigned their corresponding initial values in accordance with the rules of intrinsic assignment. Any portion of the object not identified in a DATA statement retains its status as established by preliminary initialization.

6.3.1.2.1.2 Override by initial procedure

If the type has at least one initial procedure, execution of initial procedures replaces intrinsic assignment in override initialization.

(1) If there is no explicit initialization, override initialization is performed by invoking an initial procedure for this type with an argument list that consists solely of the object being initialized, or, if no such procedure exists, by the object retaining its status as established by preliminary initialization.

(2) If the object is explicitly initialized in a type declaration statement or component definition statement, the initialization shall consist of a single defined structure constructor (4.5.6.2) for the type. Override initialization is performed by invoking an initial procedure for the type with an argument list consisting of the object itself followed by the argument list from the structure constructor.

(3) If the object is explicitly initialized in DATA statements, those initializations shall not be for component subobjects of the object. Override initialization is performed separately for each element of the object. Elements for which an initialization is provided are processed as in (2). Elements for which no initialization is provided are processed as in (1).

Proposal Note:

The above points include technical material that was not in the specification and syntax (filling in holes).

6.3.1.2.2 Override initialization of object linkage

If an object has the ALLOCATABLE attribute, explicit initialization of its linkage is prohibited. The status of the object remains as established by preliminary initialization.

If an object has the POINTER attribute, the only permitted initialization is to NULL( ). If such an explicit initialization is present, the pointer association status of the object becomes disassociated. Otherwise, the status of the object remains as established by preliminary initialization (i.e. undefined)

> Note:
>
> Since any explicit initialization for a pointer applies to its linkage, not its content, elements allocated through a pointer are always treated as having no explicit initialization for purposes of determining the nature of their override initialization.

6.3.2 How finalization is performed

As with initialization, the effect of finalization depends on whether finalization of object content or object linkage is being performed. Finalization of object content is performed for objects not having the ALLOCATABLE or POINTER attribute and for objects having one of those attributes and being deallocated in a DEALLOCATE statement. Otherwise, finalization of object linkage is performed for objects having the ALLOCATABLE or POINTER attribute.

6.3.2.1 Finalization of object content

If the type of the object is an intrinsic type, finalization of the object involves no further operations.

If the type of the object is a derived type, finalization consists of the following steps, performed in order:

(1) If the type has at least one final procedure, a final procedure is invoked with an argument list that consists solely of the object being finalized.

A standard-conforming program is required to provide for the finalization of each object that has been initialized and whose finalization would include the execution of a final procedure.

> Note:
>
> For most objects this requirement is fulfilled automatically. The effect of this requirement on a program is that it must deallocate any object it allocates through a POINTER if the type is one with a final procedure.
>
> It is not expected that processors will enforce this requirement. Rather, it allows processors, when faced with a program that violates this requirement, to process it as it wishes. Reasonable handling might be to allow the program to “leak memory” and not finalize the objects so leaked or to perform “garbage collection” to recover such objects so they can be finalized and deallocated.

(2) Each element of the object is processed as follows:

(a) Finalization is performed (recursively) for the explicitly declared component subobjects of the element, in the reverse of the order those components were declared.

(b) If the data type is an extended type, finalization is performed (recursively) for the parent component subobject of the element.

> J3 Note:
>
> See note in section 6.3.1.1.1 about parent component subobject.

6.3.2.2 Finalization of object linkage

If an object has the ALLOCATABLE attribute, finalization of its linkage consists of checking whether its allocation status is currently allocated and, if so, finalizing its content and deallocating the object.

If an object has the POINTER attribute, finalization of the object involves no further operations.

6.3.3 When initialization and finalization are performed

Initialization and finalization are performed collectively for the data objects declared in a scoping unit. Additional initialization and finalization may result from the execution of specific statements.

6.3.3.1 Initialization and finalization of scoping units

Scoping unit initialization and finalization are associated respectively with the creation and destruction of instances of the main program, subprograms, and modules.

An instance of the main program is created immediately before it is executed and destroyed when a STOP statement or the END PROGRAM statement is executed.

When a procedure defined by a subprogram is invoked from an instance of a subprogram or the main program, an instance of that subprogram is created. That instance is said to **derive** from the instance that invoked the procedure. The derived instance is destroyed when its execution completes.

If a subprogram or main program directly or indirectly references a module, each instance of the subprogram or main program accesses entities from an instance of the module determined as follows. If the instance of a subprogram derives, directly or indirectly, from an instance of a scoping unit that directly or indirectly references the same module, it accesses entities from the same instance of the module as the instance from which it derives. Otherwise, a new instance of the module is created; this new instance of the module is the one accessed by the instance of the subprogram or main program; it is not destroyed until after the instance of the subprogram or main program is destroyed.

If a subprogram or main program contains data objects in a named common block or directly or indirectly references a module containing such data objects, each instance of the subprogram or main program accesses an instance of the storage sequence for that common block determined as follows. If the instance of a subprogram derives, directly or indirectly, from an instance of a scoping unit that contains that common block or directly or indirectly references a module containing the common block, it accesses the same instance of the storage sequence as the instance from which it derives. Otherwise, a new instance of the storage sequence is created; this new instance of the storage sequence is the one accessed by the instance of the subprogram or main program; it is not destroyed until after the instance of the subprogram or main program is destroyed.

A process is permitted, at its discretion, to merge instances of modules and common block storage sequences that are not required to exist at the same time, eliminating the destruction (and associated finalization) of the earlier instance and the creation (and associated initialization) of the later instance.

> Note:
>
> Taken to the extreme, if a processor uses a strategy in which an instance of a subprogram or the main program always invokes procedures one at a time, none of the instances of modules and common block storage sequences would be required to exist at the same time, and the processor would be permitted to merge all of the instances of a given module or common block storage sequence into a single instance. Other strategies might allow merging to result in one instance per hardware processor on a multiprocessor system.

If a subprogram is nested in another scoping unit and thus has access to entities in that scoping unit by host association, an instance of the subprogram accesses those entities from the same instance of the host scoping unit from which the subprogram itself was accessed in order to be invoked.

Scoping unit initialization and finalization can be further divided into the initialization and finalization of objects that are distinct in separate instances of the scoping unit and the initialization and finalization of objects that are shared among all the instances of the scoping unit in the program.

6.3.3.1.1 Instance initialization and finalization of scoping units

Instance initialization and finalization of a scoping unit applies to all objects local to the scoping unit that are not accessed by use association or host association and that are not dummy arguments and do not have either the SAVE or the PARAMETER attribute. Instance initialization of a scoping unit also applies to dummy arguments with the INTENT(OUT) attribute, but instance finalization does not.

> Note:
>
> 6.3.3.2 provides for the corresponding actual argument to be finalized immediately before invoking the procedure, so fresh initialization is appropriate. The dummy argument is not finalized during execution of the procedure, but the corresponding actual argument is eventually finalized, so everything still "balances".

Instance initialization of a scoping unit occurs after the instance initialization of any module it references, and instance finalization occurs before the instance finalization of any referenced module.

Except during the execution of initial and final procedures, instance initialization of a nested scoping unit occurs after instance initialization of the host scoping unit and instance finalization occurs before instance finalization of the host. During execution of initial and final procedures, the instance of the host scoping unit may be only partially initialized or finalized and the program is restricted from using those objects which are not yet initialized or already finalized.

Objects in common are initialized before objects not in common and finalized after. Objects in common are not initialized if the storage sequence is not newly created. Objects in common are not finalized if the storage sequence is not about to be destroyed.

Objects in common are initialized in order of their first declaration and finalized in the reverse of that order.

Objects not in common are initialized in order of their first declaration and finalized in the reverse of that order.

A processor is permitted, as an exception to the above orderings, to initialize function result variables and INTENT(OUT) dummy arguments before the point indicated by that order and to finalize function result variables after the point indicated.

> Note:
>
> This exception allows a processor the option to put the initialization and finalization of a function result and/or the initialization of INTENT(OUT) dummy arguments in the caller of a procedure rather than the procedure itself.

> Note:
>
> The purpose of these orderings is to establish which objects can and cannot be used in initial and final procedures. Initialization and finalization may be performed in other orders as long

as the same effect is achieved. For example, it is generally possible to perform all initialization not involving initial procedures before that which does.

6.3.3.1.2 Program initialization and finalization of scoping units

Program initialization and finalization of a scoping unit applies to all objects local to the scoping unit that are not accessed by use association or host association and that have either the SAVE or the PARAMETER attribute.

Program initialization of a scoping unit occurs before all instance initialization of that scoping unit, and program finalization occurs after all instance finalization. If there is no instance initialization or finalization of a scoping unit (e.g., because a procedure is never called or a module is unreferenced), it is processor dependent whether the program initialization and finalization for that scoping unit occurs.

Note:

This allows the processor the choice of either unconditionally performing program initialization for all scoping units at the beginning of the program or of deferring program initialization of a scoping unit to the beginning of the first instance initialization for that scoping unit.

Program initialization of a scoping unit occurs after the program initialization of any module it references, and program finalization occurs before the program finalization of any referenced module.

Except during the execution of initial and final procedures, program initialization of a nested scoping unit occurs after program initialization of the host scoping unit and program finalization occurs after program finalization of the host. During execution of initial and final procedures, the program of the host scoping unit may be only partially initialized or finalized and the program is restricted from using those objects that are not yet initialized or already finalized.

Objects in common are initialized before objects not in common and finalized after. Objects in common are not initialized if the storage sequence is not newly created. Objects in common are not finalized if the storage sequence is not about to be destroyed.

Objects in common are initialized in order of their first declaration and finalized in the reverse of that order.

Objects not in common are initialized in order of their first declaration and finalized in the reverse of that order.

6.3.3.2 Initialization and finalization resulting from statement execution

Execution of an ALLOCATE statement to allocate storage for a pointer or allocatable object causes initialization of the object content. The pointer or allocatable object linkage is not initialized by this process.

Execution of a DEALLOCATE statement to deallocate storage for a pointer or allocatable object causes finalization of the object content before it is deallocated. The pointer or allocatable object linkage is not finalized by this process.

When an expression result is passed to a procedure, it is placed in an anonymous data object that is then associated with the dummy argument. Such anonymous data objects are initialized and finalized by the processor. Initialization occurs as part of the process that defines the expression result. Function result initialization is performed by the initialization of the function result variable. Defined structure constructor initialization is performed by the initial procedure (4.5.1.5) that implements that constructor. Finalization occurs at the discretion of the processor when it no longer needs that expression result. Exactly when this occurs may depend on whether a processor reuses a computed expression result or recomputes it each time it is needed.

An object or subobject, supplied as an actual argument associated with a dummy argument with the INTENT(OUT) attribute, undergoes finalization reflecting the attributes of the dummy argument immediately before the procedure is invoked. This finalization “balances” the initialization of the dummy argument during the execution of the procedure.

> Note:
>
> The attributes of the dummy argument control the nature of the finalization of the actual argument. For example, if an ALLOCATABLE object is associated with an INTENT(OUT), ALLOCATABLE dummy argument, its finalization reflects the ALLOCATABLE attribute and includes the deallocation of the object. However, if the dummy argument does not have the ALLOCATABLE attribute, the finalization applies only to the elements of the object and the object is not deallocated.

> J3 Note:
>
> It has been pointed out that the above is not the behavior called for in the TR, but that the TR was internally inconsistent in its specification of the handling of ALLOCATABLE objects and of objects of derived type with ALLOCATABLE components under these conditions, so we believe that this is the correct behavior to specify in the standard. We need to add something to section 1 to note the change from the TR.

Note:

In all cases where this finalization is non-trivial, the interface is required to be explicit, so the processor may at its discretion perform the finalization in the caller and the initialization in the procedure to minimize the amount of information that must be communicated to the procedure.

"

Secondary Edits

Proposal Note:

The first batch of secondary edits provides the rules for initial and final procedures.

42:5 Replace "*binding-name*" with "*binding-id*".

42:5+ Insert new rule and constraints: "

R439.1	*binding-id*	**is**	*binding-name*
		or	(INITIAL)
		or	(FINAL)

Constraint: The :: and the => *binding* shall not be omitted if the *binding-id* is not a *binding-name*.

"

46:35+ Insert new paragraph: "

A procedure bound to a binding identifier of (INITIAL) is an initial procedure, as described in 4.5.1.5.1. A procedure bound to a binding identifier of (FINAL) is a final procedure, as described in 4.5.1.5.2. All other type bound procedures are handled as described in this section.

"

47:20+ Insert new sections: "

4.5.1.5.1 Initial procedures

A type may have more than one initial procedure bound to it. If it does, they shall collectively satisfy the rules for unambiguous generic procedures (14.1.2.3).

Each initial procedure shall have at least one dummy argument. The first dummy argument shall be a non-optional dummy argument. It shall have the INTENT(INOUT) attribute and be of the fixed type to which the initial procedure is bound. If it is an array, it shall have assumed shape. It shall not have the ALLOCATABLE or POINTER attribute. Any additional dummy arguments shall have the INTENT(IN) attribute or shall be dummy arguments for which INTENT shall not be specified.

If the type to which the procedure is bound is extensible, the initial procedure for the type is used in the initialization process (6.3) for any extension types of that type, but it is not inherited as a initial procedure for those extension types.

If a type has any initial procedures, it shall have one whose first argument is scalar.

An initial procedure whose first argument is scalar, if it is not element (12.7), may be used **pseudo-elementally** to initialize objects of higher rank. Actual arguments conformable with the object being initialized may be associated with scalar dummy data objects not having the ALLOCATABLE or POINTER attribute. The effect of the pseudo-elemental use is to initialize the elements of the object in array element order, invoking the initial procedure for each element in sequence. Arguments to the initial procedure are as supplied, except that for conformable arguments having the same shape as the object, the corresponding element is provided. Resolution between pseudo-elemental references and references to initial procedures explicitly for higher rank is handled as for elemental procedures (12.7).

> Note:
>
> The effect of pseudo-elemental references to scalar initial procedures is similar to referencing elemental procedures, but the repetition is performed in sequence rather than in parallel and side effects are allowed.

4.5.1.5.2 Final procedures

A type may have more than one final procedure bound to it. If it does, they shall collectively satisfy the rules for unambiguous generic procedures (14.1.2.3).

Each final procedure shall have exactly one dummy argument. It shall have the INTENT(INOUT) attribute and be of the fixed type to which the final procedure is bound. If it is an array, it shall have assumed shape. It shall not have the ALLOCATABLE, POINTER, or OPTIONAL attribute.

If the type to which the procedure is bound is extensible, the final procedure for the type is used in the finalization process (6.3) for any extension types of that type, but it is not inherited as a final procedure for those extension types.

If a type has any final procedures, it shall have one with a scalar argument.

As with initial procedures, the final procedure with scalar dummy argument, if it is not elemental, may be used pseudo-elementally to finalize objects of higher rank. For final procedures, the sequence of element finalization is in the reverse of array element order.

”

> Proposal Note:
>
> The second batch of secondary edits provides for the use of structure constructor syntax to access alternative initialization by an initial procedure.

53:38-40 Replace paragraph with the following: “

A derived type implicitly defines a corresponding **structure constructor** that allows construction of values of that derived type from values of other types. A structure constructor is interpreted as an intrinsic structure constructor if the derived type has no initial procedures; it is interpreted as a defined structure constructor if the derived type has at least one initial procedure.

R448.9	*structure-constructor*	**is**	*intrinsic-structure-constructor*
		or	*defined-structure-constructor*

4.5.6.1 Intrinsic structure constructors

An intrinsic structure constructor allows a scalar value of the derived type to be constructed from a sequence of values corresponding to the components of the derived type.

”

53:41 Change “*structure-constructor*” to “*intrinsic-structure-constructor*”.

> Proposal Note:
>
> The rules for when a structure constructor is interpreted intrinsically and when it is interpreted by an initial procedure were not in the specification passed. This is my attempt to fill them in.

55:32+ Insert new subsection: “

4.5.6.2 Defined structure constructors

A defined structure constructor allows a value of the derived type to be constructed by passing arguments to an initial procedure for the type.

R450.1	*defined-structure-constructor*	**is**	*derived-type-spec* ([*argument-list*])

Constraint: The *argument-list* shall be a valid specification of actual arguments corresponding to the dummy arguments other than the first for one of the initial procedures for the type.

If a defined structure constructor is used as an explicit initialization (5.1), its argument list is used in the initialization process (6.3) for the object to which the explicit initialization applies.

If a defined structure constructor is in any other context, the argument list is used in the initialization of an anonymous object of that type. If the argument list is valid in the initialization process for objects of more than one possible rank, the anonymous object has the minimum such rank. The anonymous object is the "value" of the structure constructor.

"

Proposal Note:

The current syntax for declaring extension types provides no means for the extension type to specify a different default initialization for its "parent component" or link alternative initialization of the extension type to alternative initialization of the parent. One can work around this by simply allowing the parent component to be initialized "wrong" and overriding that initialization in an initial procedure for the extension, but this is a bit awkward and causes the processor to do more work during the initialization process. This is not something I can fix in the proposal, but it might be something we should think about.

Related Minor Edits

39:10-13 Replace the first three sentences of the paragraph with "Initialization in a component declaration specifies **default initialization** for that component. This default initialization contributes to the initialization process (6.3) for objects of that type."

Proposal Note

Many of the details here have been incorporated into 6.3.

Sometimes we talk about default initialization and sometimes we talk about component initialization. Is there a consistent reason for using one term or the other?

44:17 Replace "is initially … (14.7.5)" with "is default initialized (6.3)"

44:18-25 Delete sentences after first two.

Proposal Note:

This material is covered in another way in 6.3.

70:42-44 Replace second sentence in the paragraph with "On invocation of the procedure, such a dummy argument undergoes the initialization process (6.3)."

> Proposal Note:
>
> The new complications are all described in 6.3.

72:17-18 Replace the final sentence in the paragraph with "Because an INTENT(OUT) variable has no access to any part of the previous status of the actual argument, the initialization process is applied to it."

75:11-19 Replace the first two paragraphs of the section:

"If an object has the **SAVE attribute**, a single copy of the object is shared among all instances (6.3.3.1, 12.5.2.3) of the scoping unit in which it appears, allowing association status, allocation status, definition status, and value established in one instance to be referenced in another instance. Such an object is called a **saved object**.

If an object in an executable scoping unit does not have the SAVE attribute, a separate copy of that object exists for each instance of the scoping unit. For objects in a module that do not have the SAVE attribute, the basis for sharing is that the instance of a module referenced by an executable scoping unit is the one available to all instances resulting from its procedure invocations. Thus, two instances of executable scoping units referencing a module share the same instance if and only if one is the direct or indirect result of the other or both are the direct or indirect result of a third instance that also references that module."

> Proposal Note:
>
> In FORTRAN 77, the descriptive model was that variables existed for the life of a program, but that in the absence of a SAVE statement, they became undefined between executions. In Fortran 90, in order to accommodate features such as recursion and automatic array, we switched to the model that in the absence of SAVE, each execution had its own copy of the variable. However, many vestiges of the F77 descriptive model remain. Because the semantics of initial and final procedures are strongly tied to the F90 model, it is helpful to revise some of the text that still reflects the F77 model.
>
> The pointer part of this is handled elsewhere.

81:38-42 Replace the second sentence in the paragraph:

"For a common block declared in a SAVE statement, its common block storage sequence (5.6.2.1) in an instance of the scoping unit containing the common block is associated with the common block storage sequence for every other instance of a scoping unit containing the common block, thus making the values in those sequences available to all instances. For a common block not having the SAVE attribute, the basis for sharing is that the common block storage sequence for a common block in an instance of an executable scoping unit is associated with all such common block storage sequences in instances resulting from its procedure invocations. Thus, the common block storage sequences in two instances of executable scoping units containing a common block are associated if and only if one instance is the direct or indirect result of the other or both are the direct or indirect result of a third instance that also contains that common block."

93:10+ Insert note: "

> Note:
>
> A *common-block-object* must not be of a type with initial or final procedures, since initial and final procedures are type-bound procedures and sequence types do not have type-bound procedures.

"

> Proposal Note:
>
> Yes, I really mean "must" here. This is an implied requirement, not a direct one.

107:20-43 Delete.

> Proposal Note:
>
> This should be covered in 6.3.

108:2 Insert new sentence: "The effect of terminating execution of a procedure on an allocatable object accessed by use association is affected by the presence or absence of the SAVE attribute and by the processor-dependent merging of instance of modules (6.3.3)."

110:1-20 Delete.

> Proposal Note:
>
> This should be covered in 6.3.

110:22 Insert new sentence: "Processor merging of instances of modules (6.3.3.1) can make it processor-dependent whether an allocatable object accessed by use association is deallocated when execution of a procedure is terminated."

110:44-111:11 Delete.

Proposal Note:

This should be covered in 6.3

245:23+ Insert new item into list: "

(d) A dummy argument that has the INTENT(OUT) attribute and is of a type whose finalization process (6.3) includes the execution of a final procedure,

"

Proposal Note:

This might have been missing from the version of the specification that was passed.

258:6-7 Replace final sentence in paragraph with sentence as for 72:17-18 above.

259:34-36 Replace second sentence with "No initialization (6.3) is performed on the dummy data object, even if it has INTENT(OUT)."

269:46+ Insert note: "

Note:

This includes any initial or final procedure invoked in the initialization process for data objects local to the procedure.

"

346:33-41 Replace both items with "(4) The pointer becomes disassociated by the initialization process (6.3)."

347:1-5 Replace all three items with "(2) The target of the pointer is finalized (6.3) by means other than deallocating the pointer."

Proposal Note:

This text covers both (2) and (3). (4) is covered by the pointer ceasing to exist, so it doesn't matter if it becomes undefined; the new instance of the pointer will be undefined anyway.]

350:13-19 Combine items (1)-(3) into "(1) Variables that are defined by the initialization process (6.3)."

351:29-30 Replace item with "(16) The initialization process (6.3) may cause part or all of an object to become defined."

351:35-45 Delete.

> Proposal Note:
>
> Subsumed by the above.

352:15-28 Delete.

> Proposal Note:
>
> These are the variables that cease to exist.

353:4-5 Replace item with "(11) Successful execution of an ALLOCATE statement for a nonzero-sized object causes the object to become undefined unless the initialization process (6.3) defines it."

353:12-13 Replace "except … specified" with "unless is it defined by the initialization process (6.3)"

353:14-15 Delete.

> Proposal Note:
>
> This case is now covered by the fact that the actual argument is finalized and not usable until it has been reinitialized through the dummy argument.

353:19-20 Same replacement as for 353:12-13.

•