This paper elaborates on the preliminary specification presented in J3/97-209r1. The rationale and overall strategy is unchanged and will not be repeated here. Some of the questions raised in that paper are answered here, but the remainder should be considered possible areas for future action even though they are not repeated here.

To simplify the description of the interaction between initial procedures and other initialization features and between final procedures and other automatic "clean up" features, the entire initialization and finalization processes are described here, including the aspects that are already part of the Fortran language.

Basic Concepts

Initialization is the process that takes place between the time a data object comes into existence (i.e., in a very general sense, the time it is allocated) and the first time it is available for explicit use. Finalization is the process that takes place between the last time it is available for explicit use and the time is ceases to exist (i.e., in a very general sense, the time it is deallocated).

Initialization takes place under the following circumstances:

1. For each scoping unit, there is "per program" initialiazation that takes place once per execution of the program.

2. For each scoping unit, there is "per instance" initialization that takes place once for each instance of the scoping unit that comes into existence. E.g., each time a procedure is invoked, the per instance initialization of its defining subprogram takes place.

3. Initialization takes place during the execution of an ALLOCATE statement for the objects allocated.

Finalization takes place under the following circumstances:

1. For each scoping unit, there is a "per program" finalization that takes place once per execution of the program.

2. For each scoping unit, there is a "per instance" finalization that takes place once for each instance of the scoping unit that comes into existence. E.g., each time an invocation of a procedure completes, the per instance finalization of its defining subprogram takes place.

3. Finalization takes place during the execution of a DEALLOCATE statement for the objects deallocated.

4. When an object is associated with an INTENT(OUT) dummy argument, it is finalized. (In effect, this prepares it to receive the result of the initialization of the

dummy argument.) Note that the association of a subobject, such as an array section, with an INTENT(OUT) dummy argument causes that subobject to be finalized and re-initialized while the remainder of the object is unaffected!

Effect of Attributes on Initialization

An object with the SAVE attribute is initialized as part of the per program initialization for the scoping unit of which it is a part and finalized as part of the per program finalization.

An object with the PARAMETER attribute is initialized as part of the per program initialization for the scoping unit of which it is a part and finalized as part of the per program finalization.

With one exception, a dummy argument is neither initialized nor finalized (depending, instead, on the normal initialization and finalization of the associated actual argument). The exception is for a dummy argument with INTENT(OUT). It is initialized but not finalized, with finalization of the corresponding actual argument taking place before the initialization of the dummy argument. [As a model for this behavior, one might imagine that the actual argument is deallocated (in the general sense), triggering the finalization, and then re-allocated through the dummy argument, triggering the initialization].

An object not covered by the preceding paragraphs is initialized as part of the per instance initialization of the scoping unit of which it is a part and finalized as part of the per instance finalization.

The *per program* or *per instance* initialization of an object with the POINTER attribute is simply that its pointer association status is undefined (if there is no explicit initialization) or disassociated (if there is an explicit initialization to NULL). Finalization of an object with the POINTER attribute has no effect. Note, however, that the execution of an ALLOCATE or DEALLOCATE statement containing an object with the POINTER attribute causes initialization or finalization of the target object. (Pointer assignment does not trigger initialization or finalization.) [If procedure pointers are not considered data objects, the relevant portion of the above rules may have to be repeated for them.]

The *per program* or *per instance* initialization of an object with the ALLOCATABLE attribute is simply that its allocation status is “not allocated”. Finalization of an object with the ALLOCATABLE attribute results in the object being deallocated if it is allocated. This deallocation triggers the same finalization as a DEALLOCATE statement. Execution of an ALLOCATE or DEALLOCATE statement containing an object with the ALLOCATABLE attribute causes the initialization or finalization that would have occurred on a per program or per instance basis if the ALLOCATABLE attribute were not present.

Initialization/Finalization Order

The per program initialization for a scoping unit precedes all per instance initialization for that scoping unit, and the per program finalization for a scoping unit follows all per instance finalization for that scoping unit.

If a scoping unit references a module, the module's per program initialization will precede the scoping unit's per program initialization, the module's per program finalization will follow the scoping unit's per program finalization, the module's per instance initializations will precede the scoping unit's per instance initializations for corresponding instances, and the module's per instance finalizations will follow the scoping unit's per instance finalizations for corresponding instances.

If a scoping unit is contained in a host scoping unit, the host's per program initialization will precede the scoping unit's per program initialization, the host's per program finalization will follow the scoping unit's per program finalization, the host's per instance initializations will precede the scoping unit's per instance initializations for corresponding instances, and the host's per instance finalizations will follow the scoping unit's per instance finalizations for corresponding instances.

If a scoping unit contains the definition of a COMMON block, the per program initialization (if any) of objects in that COMMON block will precede the per program initialization of objects not in COMMON, the per program finalization (if any) of objects in that COMMON block will follow the scoping unit's per program finalization of objects not in COMMON, the per instance initializations (if any) of objects in that COMMON block will precede the scoping unit's per instance initializations of objects not in COMMON for corresponding instances, and the per instance finalizations (if any) of objects in that COMMON block will follow the scoping unit's per instance finalizations of objects not in COMMON for corresponding instances.

Function result variables and INTENT(OUT) dummy arguments may be initialized prior to all other objects in the per instance initialization for the scoping unit and shall occur no later than the point at which they would be initialized if they were not in this category. Similarly, a function result variable may be finalized after all other objects in the per instance finalization for the scoping unit (and typically will be to allow the function result to be used before the result temporary is finalized) and shall occur no sooner than the point at which it would be finalized if it were not a function result variable.

If two objects are both initialized in the per program initialization for a scoping unit and their order of initialization is not constrained by the above rules, they shall be initialized in the order of their first appearance in that scoping unit's declarations. If two objects are both finalized in the per program finalization for a scoping unit and their order of finalization is not constrained by the above rules, they shall be finalized in the reverse order. If two objects are both initialized in a per instance initialization for a scoping unit and their order of initialization is not constrained by the above rules, they shall be initialized in the order of their first appearance in that scoping unit's declarations. If two

objects are both finalized in a per instance finalization for a scoping unit and their order of finalization is not constrained by the above rules, they shall be finalized.

Breaking Down the Process

Initialization of an object of intrinsic type results in the object having an undefined value (if no explicit initialization is specified) or having a specific defined value (if explicit initialization is specified.

Finalization of an object of intrinsic type has no effect.

Initialization of an object of derived type consists of the following steps:

1. If the type is an extension of a parent type, the subobject of that parent type is initialized (applying these steps recursively).

2. The components are initialized in order (applying these steps recursively). If a component has been declared with a default initialization, it is treated like an explicit initialization of that component during the recursive application of these steps.

3. One of the following occurs:

   a. If the object has an explicit initialization declared and the type has no initial procedures bound to it, the explicit initialization is applied to the object.

   b. If the type has an initial procedure bound to it and the object has no explicit initialization declared, the initial procedure is executed for that object. (The initial procedure would be one whose only argument is the object.)

   c. If the object has an explicit initialization declared and the type has initial procedures bound to it, then the explicit initialization shall correspond to "extra" argument(s) on one of the initial procedures, and that initial procedure is executed with both the object and the explicit initialization as arguments.

   d. If the object does not have an explicit initialization declared and the type has no initial procedures bound to it, no action is taken at this step.

[This ordered application of initialization steps (in which later steps effectively override the actions of earlier steps) replaces the rules in F95 that dictate which initializations take precedence. The final result should be the same.]

Finalization of an object of derived type consists of the following steps:

1. If the type of the object has a final procedure bound to it, it is executed for the object.

2. The declared components of the object are finalized in reverse order (applying these steps recursively).

3. If the type is an extension of a parent type, the subobject of that parent type is finalized (applying these steps recursively).

Miscellaneous Details

The current limitations on default initialization in COMMON would be retained. Rather than trying to formulate comparable rules for objects with initial or final procedures, we are proposing the simpler rules of disallowing such objects in COMMON.

All per program finalization occurs before the automatic closing of open input/output units at program termination.

Normal termination, including the execution of a "plain" STOP statement, will normally trigger the per instance finalization for currently active instances of scoping units and the per program finalization of all scoping units. There should be some form of termination that can be used in error conditions to bypass this work and terminate quickly.

Any object that is of a type with a final procedure and is ALLOCATEd shall be DEALLOCATEd by the program. (This is intended to give implementations the freedom, when presented with a program that violates this restriction, of either garbage collecting the object and performing the finalization for it or not doing so and missing that finalization process.)

If two instances of a module or COMMON block never exist at the same time, the processor is permitted to combine them into a single instance, effectively suppressing one execution of the finalization and one execution of the initialization. If none of the instances of a module or COMMON block overlap in time, repeated applications of this rule would allow reduction of those instances to a single instance for the entire program, with the first per instance initialization and last per instance finalization becoming an extension of the per program initialization and finalization. (Most processors treat unSAVEd COMMON blocks and unSAVEd objects in modules no differently from SAVEd COMMON blocks and SAVEd objects in modules. This rule can be used to allow a processor to continue that practice.)

To avoid the necessity of having separate initial procedures for scalar objects, 1-dimensional objects, 2-dimensional object, … without imposing the restrictions of an elemental procedure, the processor shall, in the absence of array versions of the initial procedure apply the scalar version to the elements of an array in sequence order.

Preliminary Syntax

Initial and final procedures would be identified by making them specially-identified type-bound procedures (e.g. with the "names" like (INITIAL) and (FINAL) [including the parentheses]). (Use of the type-bound procedure mechanism gets us several desirable rules for free, including the exclusion of SEQUENCE types and the guarantee that they will be available wherever the type is, independent of how the programmer has managed names. Its suitability for this purpose is, however, dependent on some other details of this feature that have not yet been approved. The backup position would be to use (INITIAL) and (FINAL) as *generic-spec*s and express those "free" rules explicitly.)

The first dummy argument of these type-bound procedures would be of that type and have INTENT(INOUT). A finalization procedure would have no additional arguments. An initialization procedure would be permitted additional INTENT(IN) arguments for use when explicit initialization is provided. For example, if

```
SUBROUTINE myreal_init(object,value)
  TYPE(myreal),INTENT(INOUT)::object
  REAL :: value
  …
END SUBROUTINE myreal_init
```

is made an initial procedure, this would allow a declaration of the form

```
TYPE(myreal)::myvar=1.0
```

The syntax for the multi-value case is still being explored. Perhaps

```
SUBROUTINE mycomplex_init(object,value1,value2)
  TYPE(mycomplex),INTENT(INOUT)::object
  REAL :: value1,value2
  …
END SUBROUTINE mycomplex_init
```

would allow

```
TYPE(mycomplex)::myvar=mycomplex(0.0,1.0)
```

It is unclear how this syntax would extend to arrays. (Pseudo-elementally?)

Ω