Date: 4 March 1999
To: J3
From: Van Snyder
Subject: Unresolved issues 16, 72, 74-76, 78, 79, and 81 concerning explicitly typed allocations
References: 98-208r2

Except for changes to address unresolved item 72, the reason for each change is indicated in the margin, along with the position to which an edit applies (the great majority of changes apply to item 72).

The decision for each unresolved issue was:

16 Noticed that some of it is already covered elsewhere. Fixed the rest as a by-product of plugging a hole noted below (deferred parameters of function result types).

72 Grepped for all the instances of "allocate" (as I should have before writing 98-208). Added another "ALLOCATED" intrinsic with an argument named SCALAR that applies to scalars.

74 The requirement to allow allocatable and pointer arrays not to have deferred shape is withdrawn because it introduces undesirable opportunities to get run-time errors that are not compile-time checkable. **spec change**

75 Introduced three constraints to specify what's compile- time checkable, and specified what is required to be run-time checked, and which if it fails thereby causes an error status to be returned.

76 It is necessary to specify all of the type parameters in an ALLOCATE statement if any are specified. It's necessary to specify them if any are deferred in the declaration. **spec change**

78 Clarified the wording. Doing so required thinking about functions results that have deferred type parameters. This is mentioned in the next paragraph.

79 Clarified the wording. The result is that if a dummy argument has deferred type parameters, the corresponding actual argument shall have deferred the same type parameters, and that the non-deferred type parameters of actual and dummy arguments shall have the same values.

81 Did what the editor suggested.

Two holes are plugged:

- The behavior of function result types that have deferred parameters is defined.
- The behavior of assumed type parameters during allocation is defined.

1 Edits

Edits refer to 99-007. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by +

indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: In the same paragraph.] New in Fortran 2000 is the ability to declare that structure components and objects that are not arrays have the ALLOCATABLE attribute; they therefore are automatically deallocated under the same conditions as allocatable arrays were automatically deallocated in Fortran 95. Also new in Fortran 2000 is the capability to specify nonkind type parameters, and the specific types of polymorphic objects, during allocation. — xvi:21+

[Editor: Delete from "A deferred" to the end of the text of unresolved issue 81. The sentence on 31:1-2 is moved to 100:13+. Add the following:] — 31:1-8 81

If a colon is specified for the *type-param-value* of a nonkind type parameter, the type parameter is a deferred type parameter. The value of a deferred type parameter of an object may be specified when the object is allocated (6.4.1), assumed from an actual argument when the object is a dummy argument associated with an actual argument, or assumed from a *target* during execution of a pointer assignment statement (7.5.2) in which the object appears as a *pointer-object*.

[Editor: Replace "a pointer or allocatable array" by "allocatable or a pointer"] — 38:28

[Editor: Replace "allocatable arrays or pointers" by "allocatable or pointers"] — 38:36

[Editor: Delete "an" and "array"] — 45:37

[Editor: Delete "arrays" twice] — 49:4,6

Constraint: If an asterisk is used as a *type-param-value* in the declaration of an entity, it shall be a dummy argument. — 53:23-26

Constraint: If a colon is used as a *type-param-value* in the declaration of an entity or component, it shall have the ALLOCATABLE or POINTER attribute.

Constraint: When a *type-param-value* appears other than within an ALLOCATE statement, the *scalar-int-expr* shall be a *specification-expr*.

[Editor: Delete "an" and "array"] — 55:19

[Editor: Replace "array" by "entity"] — 55:20

[Editor: Replace "shall evaluate to array" by "an entity of the same rank"] — 55:21

[Editor: Replace "array" by "entity" twice.'] — 55:23,24

[Editor: Replace "shape" by "bounds" twice.] — 55:24,26

[Editor: Delete "array".] — 55:28

[Editor: Replace "an allocatable array" by "allocatable"] — 55:230

[Editor: Delete (if you agree this paper fixes issue 72).] — 63:1-33

[Editor: Replace "array" by "variable"] — 63:43

[Editor: Replace "an allocatable array" by "allocatable"] — 63:43-44

[Editor: Replace "array" by "variable"] — 64:31

[Editor: Delete "an" and "array"] — 64:32

[Editor: Delete "an" and "array"] — 64:37

Constraint: When a *type-value* appears other than within an ALLOCATE statement, the *scalar-int-expr* shall be a specification expression. — 68:4+

[Editor: Insert into the list:]	68:20+
($3\frac{1}{2}$) It may be used in an ALLOCATE statement to denote the assumed length of a dummy argument.	
Constraint: If a *proc-entity* is an external function, *proc-interface* is present, and the result type is not character, the type parameters of the result type shall be deferred or specified by initialization expressions. If the result type is character, the length parameter shall be deferred, specified by an initialization expression, or an asterisk.	79:2+ 16, deferred parms of function results
[Editor: Delete unresolved issue 16. The first paragraph is covered by 246:47-48, which applies to abstract interfaces, and the second is fixed below (I hope). I couldn't find where there is a definition for what is meant by "the procedure's characteristics shall be consistent with those specified in the procedure definition" at 246:47-48, but when I do find it I presume there will be work to be done there to account for deferred parameters.]	79:11-27 16, deferred parms of function results
If *proc-interface* is present and consists of *abstract-interface-name*, it specifies an explicit specific interface (12.3.2.1) for the declared procedures or procedure pointers. If *proc-interface* is present and consists of *declaration-type-spec*, it specifies that the declared procedures or procedure pointers are functions having implicit interface and the specified result type. If a type is specified for an external function, its function definition (12.5.2.1) shall specify the same result type and type parameters. If *proc-interface* is absent, the procedure declaration statement does not specify whether the declared procedures or procedure pointers are subroutines or functions. Deferred parameters of function result types have no values; they simply indicate that those parameters of the function result will be determined by the function, when it is invoked.	79:33+ 16, deferred parms of function results
[Editor: Change "array" to "variable"]	84:40
[Editor: Remove "array as an"]	84:44
[Editor: Change "array" to "variable"]	89:41
[Editor: Remove "an" and "array"]	89:42
[Editor: Change "array" to "variable"]	90:31
[Remove "array as an"]	90:32
[Change "array" to "variable"]	93:1
[Change "an allocatable array" to "allocatable"]	93:2
[Change "arrays" to "variables"]	97:26
A deferred type parameter of a disassociated pointer, a function procedure pointer, an unallocated variable, or a pointer with undefined association status shall not be referenced.	100:13+ 81
[Change "arrays" to "variables" twice.]	104:12,17
[Editor: Delete – covered by 104:28-29.]	104:33-34
Constraint: If an *allocate-object* has a deferred type parameter, *type-spec* shall appear. Constraint: A *type-param-value* in the *type-spec* shall not be a colon. Constraint: The *type-param-value* shall be asterisk if and only if each *allocate-object* is a dummy argument for which the corresponding type parameter is assumed.	74 104:34+ 75
[Editor: Delete (if you agree this paper fixes lines 105:3-5 of issue 74; lines 6-10 are updated	105:1-10 74

and moved downward).]

When an ALLOCATE statement having a *type-spec* is executed, type parameters are specified by *type-param-values* in the *type-spec* in the ALLOCATE statement. If the value specified for a nondeferred type parameter is not the same as the value specified in the object's declaration, an error condition occurs. 105:17-40 74 76

If a *type-param-value* in a *type-spec* in an ALLOCATE statement is an asterisk it denotes the current value of that assumed type parameter.

For derived types, values are assigned to type parameters as specified in 4.5.5. For intrinsic types, values are assigned to type parameters, with the correspondence determined by position or by name as defined in 5.1.

[Editor: Replace "array" by "entity"] 107:11

[Editor: Replace "array" by "variable" every time it appears in this range (and replace "an" by "a" as necessary). Yes, I checked them all.] 107:13-43

[Editor: Replace "array" by "variable" twice.] 109:13,16

[Editor: Replace "array" by "variable" every place it appears in this range (and replace "an" by "a" as necessary). Yes, I checked them all.] 109:37-110:11

[Editor: Replace "an allocatable array" by "allocatable" the first two times, remove the last "array".] 110:15-16

[Editor: Replace "allocated allocatable arrays" by "allocatable and currently allocated"] 110:17

[Editor: Replace "an array" by "a variable"] 110:22

[Editor: Replace "array" by "entity"] 110:38

[Editor: Remove "array."] 119:41

[Editor: Replace "array" by "entity"] 119:42

[Editor: Remove the word "array" twice.] 138:38-39

[Editor: Remove the word "array"] 139:13

Constraint: If the *pointer-object* is a function procedure pointer, the *pointer-object* and *target* shall have deferred corresponding type parameters. 140:19+ 78

[Editor: Delete. It's covered by the constraint at 140:16-17.] 140:24-26

If *pointer-object* is a data object or function procedure pointer, all nondeferred type parameters of *pointer-object* shall have the same values as corresponding type parameters of *target*. If *pointer-object* is a data object that has deferred type parameters, the values of those parameters are assumed from the values of corresponding parameters of *target*. The corresponding parameters of *target* can be deferred, assumed, or explicitly declared. 78 141:11-19 78

Deferred parameters of function procedure pointers have no values; instead, they indicate that those parameters of the function result will be determined by the function, when it is invoked. Remember that any entity with deferred type parameters, including a function result, is required to have the ALLOCATABLE or POINTER attribute. Note 7.46½

Remove unresolved issue 78. There is no constraint that all of the type parameters agree, only that all kind type parameters agree; kind type parameters cannot be deferred. The paragraph at 141:11- 13 *does* discuss dynamic type parameters, by saying that values of deferred type parameters of *pointer-object* are assumed by *target*. Verifying that the values of nondeferred nonkind type parameters of *pointer-object* are the same as corresponding parameters of *target* requires a run-time check, regardless whether nondeferred parameters of *pointer-object* correspond to deferred or nondeferred parameters of *target*. The most obvious case is when they're both assumed. If the check fails there's no way to catch it. This is in the same category as array bounds checking. A subscript out-of-bounds is an indication that the program is not standard-conforming; so is a mismatch of *pointer-object* and *target* type parameters. I know lots of people want to use square brackets for array constructors, but I have another suggestion, at least within pointer assignment statements: Let me put [STAT=*variable* and/or ERRMSG=*default-char-variable*] at the end of the statement, so that I can catch mismatched type parameters.	J3 note (not an edit)
[Editor: Replace "an allocatable array" by "allocatable"]	185:14
[Editor: Remove the word "array"]	185:35
[Editor: Replace "an allocatable array" by "allocatable"]	244:14
[Editor: Replace "a pointer or an allocatable array" by "allocatable or a pointer" twice.]	244:26-27
(b) A dummy argument that is allocatable, an assumed shape array, a pointer, or a target,	245:21-23 72 76
The constraint at line 23 is not needed, because of the constraint at 53:25-26.	FYI
Except for the case of the character length parameter of an actual argument of type default character associated with a dummy argument that is not assumed shape, the type parameters of an actual argument that correspond to nondeferred nonassumed type parameters of the associated dummy argument shall have the same values as corresponding type parameters of the associated dummy argument.	254:29-31 79
In the most generally allowed case, verifying that the value of a nondeferred nonassumed type parameter of a dummy argument is the same as the corresponding type parameter of the associated actual argument requires a run-time check. If the check fails, there's no way to trap it. This is in the same category as array bounds checking. A subscript out-of-bounds is an indication that the program is not standard-conforming; so is a mismatch of actual and dummy argument type parameters. Should this remark be in appendix C, or is it enough just to have it transiently for ourselves?	J3 note (not an edit)
If a dummy argument that does not have INTENT(OUT) has deferred or assumed type parameters, the initial values of those parameters are assumed from the values of the corresponding type parameters of the associated actual argument.	
If the dummy argument that does not have INTENT(IN) has deferred type parameters (and is therefore allocatable or a pointer), it may be re-allocated using type parameter values different from the original deferred parameter values of the associated actual argument. If it is a pointer, it may acquire type parameter values different from the original deferred parameter values of the associated actual argument by pointer assignment.	Note 12.19$\frac{1}{2}$
An actual argument associated with a dummy argument that is allocatable or a pointer shall have deferred the same type parameters as the dummy argument.	
[Editor: Delete]	255:15-31 79

The edits at 254:29-31 make it clear that deferred parameters *can* be changed if the dummy argument doesn't have INTENT(IN). The intent *is* that actual and dummy arguments shall have deferred the same type parameters – even in the INTENT(IN) case. If you really want to use deferred parameters just to get at the values of the actual argument's parameters, you should use assumed parameters instead. This is not precisely symmetrical to the case of pointer assignment because when you do a pointer assignment, and then later re-allocate either the *pointer object* or the *target*, the other one isn't affected. This is not true in the case of actual and dummy arguments – if you re-allocate the dummy argument, the associated actual argument gets reallocated, so it better have deferred at least all of the deferred parameters of the dummy argument. Specifying concrete values for parameters of the dummy that correspond to deferred parameters of the actual is just an opportunity for them to disagree. If you want to enforce a specific value for a parameter of the dummy that corresponds to a deferred parameter of the actual, defer the dummy's parameter, and check its value explicitly. If you want to set a deferred parameter of the actual argument to a specific value during allocation, defer the corresponding parameter of the dummy and put the desired value in the allocate statement, not the type declaration for the dummy argument. Should this remark be in appendix C, or is it enough just to have it transiently for ourselves?	J3 note (not an edit)
[Editor: Replace "an allocatable array" by "allocatable" twice.]	257:18
[Editor: Remove the first sentence.]	277:32-33
[Editor: Add a new section] **13.9 Allocation status inquiry functions** The inquiry function ALLOCATED tests whether an allocatable variable is currently allocated.	278:15+
[Editor: Delete this line. Moved to 283:9+.]	282:31
[Editor: Add a new section] **13.12.20 Allocation status inquiry functions** ALLOCATED (ARRAY) Array allocation status ALLOCATED (SCALAR) Scalar variable allocation status	283:9+
[Editor: Add a new section] **13.15.10 ALLOCATED (SCALAR)** **Description.** Indicate whether or not an allocatable scalar is allocated. **Class.** Inquiry function. **Argument.** SCALAR shall be an allocatable scalar. **Result Characteristics.** Default logical scalar. **Result Value.** The result has the value true if SCALAR is currently allocated and has the value false if SCALAR is not currently allocated.	287:39+
[Editor: Replace "or ... allocated" by ". If it is allocatable it shall be currently allocated."]	306:13
[Editor: Add the sentence "If it is an unallocated allocatable variable, it shall not have deferred length."]	306:35
[Editor: Replace "array" by "object"]	318:15
[Editor: Remove the first "array" and replace the second one by "object"]	318:23
[Editor: Replace "array" by "entity"]	326:32
(5) An *object-name* in an *allocate-stmt*;	343:38

($5\frac{1}{2}$) An *array-name* in a *dimension-stmt*;

[Editor: Replace "array" by "entity"] 353:3

---

allocatable variable (5.1.2.4.3): A *variable* having the ALLOCATABLE *attribute*. It may be *referenced* or *defined* only when it has space allocated. If it is an *array*, it has a *shape* only when it has space allocated. It may be a *named variable* or a *structure component*. 385:12-14

[Editor: Replace "array" by "variable"] 387:31

[Editor: Replace "a pointer or allocatable array" by "allocatable or a pointer"] 388:4

C.11.1.4 Automatic arrays and allocatable variables (5.1, 5.1.2.4.3) 436:1-10

A major advance for writing modular software is the presence of automatic arrays, created on entry to a subprogram and destroyed on return, and allocatable variables, including arrays whose rank is fixed but whose actual size and lifetime is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE statements. The declarations

```
SUBROUTINE X (N, A, B)
...
REAL WORK (N, N); REAL, ALLOCATABLE :: HEAP (:, :)
```

specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate storage mechanism for the implementation of automatic arrays, but a heap will be needed for allocatable variables.