ISO/IEC Technical Report

ISO/IEC JTC1 PDTR 20.02.01.04

Enhanced Derived Type Facilities

in

Fortran

An extension to IS 1539-1 : 1996

{Produced 2-Jul-96}

THIS PAGE TO BE REPLACED BY ISO CS

Contents

1. GENERAL 1

1.1 Scope 1

1.2 Normative References 1

2. RATIONALE 2

3. REQUIREMENTS 3

3.1 Allocatable Attribute Regularization 3

3.2 Allocatable Arrays as Dummy Arguments 3

3.3 Allocatable Array Function Results 4

3.4 Allocatable Array Components 5

4. REQUIRED EDITORIAL CHANGES TO ISO/IEC 1539-1 : 1996 8

Foreword

[This page to be provided by ISO CS]

Introduction

This technical report defines a proposed extension to the data-typing facilities of the programming language Fortran. The current Fortran language is defined by the international standard ISO/IEC 1539-1 : 1996. This technical report has been prepared by ISO/IEC JTC1/SC22/WG5, the technical working group for the Fortran language. The language extension defined by this technical report is intended to be incorporated in the next revision of the Fortran language without change except where experience in implementation and usage indicates that changes are essential. Such changes will only be made where serious errors in the definition or difficulties in integration with other new facilities are encountered.

This extension is being defined by means of a technical report in the first instance to allow early publication of the proposed definition. This is to encourage early implementation of important extended functionalities in a consistent manner and will allow extensive testing of the design of the extended functionality prior to its incorporation into the language by way of the revision of the international standard.

Information technology - Programming Languages - Fortran Technical Report: Data-type enhancements

1. General

1.1 Scope

This technical report defines a proposed extension to the data-typing facilities of the programming language Fortran. The current Fortran language is defined by the international standard ISO/IEC 1539-1 : 1996. The proposed extension allows components of derived types to be allocatable arrays.

Section 2 of this technical report contains a general informal but precise description of the proposed extended functionalities. This is followed by detailed editorial changes which if applied to the current international standard would implement the revised language definitions.

1.2 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this technical report. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this technical report are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 1539-1 : 1996 *Information technology - Programming Languages - Fortran*

2. Rationale

There are many situations when programming in Fortran where it is necessary to allocate and deallocate arrays of variable size but the full power of pointer arrays is unnecessary and undesirable. In such situations the abilities of a pointer array to alias other arrays and to have non-unit (and variable at execution time) strides are unnecessary, and they are undesirable because this limits optimization, increases the complexity of the program, and increases the likelihood of memory leakage. The ALLOCATABLE attribute solves this problem but can currently only be used for locally stored arrays, a very significant limitation. The most pressing need is for this to be extended to array components; without allocatable array components it is overwhelmingly difficult to create opaque data types with a size that varies at runtime without serious performance penalties and memory leaks.

A major reason for extending the ALLOCATABLE attribute to include dummy arguments and function results is to avoid introducing further irregularities into the language. Furthermore, allocatable dummy arguments improve the ability to hide inessential details during problem decomposition by allowing the allocation and deallocation to occur in called subprograms, which is often the most natural position. Allocatable function results ease the task of creating array functions whose shape is not determined initially on function entry, without negatively impacting performance.

3. Requirements

The following subsections contain a general description of the extensions required to the syntax and semantics of the current Fortran language to provide facilities for regularization of the ALLOCATABLE attribute.

3.1 Allocatable Attribute Regularization

In order to avoid irregularities in the language, the ALLOCATABLE attribute needs to be allowed for all data entities for which it makes sense. Thus, this attribute which was previously limited to locally stored array variables is now allowed on

- array components of structures,
- dummy arrays, and
- array function results.

Allocatable entities remain forbidden from occurring in all places where they may be storage-associated (COMMON blocks and EQUIVALENCE statements). Allocatable array components may appear in SEQUENCE types, but objects of such types are then prohibited from COMMON and EQUIVALENCE.

The semantics for the allocation status of an allocatable entity remain unchanged:

- If it is in a main program or has the SAVE attribute, it has an initial allocation status of not currently allocated. Its allocation status changes only as a result of ALLOCATE and DEALLOCATE statements.
- If it is a module variable without the SAVE attribute, the initial allocation status is not currently allocated and the allocation status may become not currently allocated (by automatic deallocation) whenever execution of a RETURN or END statement results in no active procedure having access to the module.
- If it is a local variable (not accessed by use association) and does not have the SAVE attribute, the allocation status becomes not currently allocated on entry to the outermost procedure which has access to it. On exit from this procedure it is automatically deallocated and the allocation status changes to not currently allocated.

Since an allocatable entity cannot be an alias for an array section (unlike pointers arrays), it may always be stored contiguously.

3.2 Allocatable Arrays as Dummy Arguments

An allocatable dummy argument array shall have associated with it an actual argument which is also an allocatable array.

On procedure entry the allocation status of an allocatable dummy array becomes that of the associated actual argument. If the dummy argument is not INTENT(OUT) and the actual argument is currently allocated, the value of the dummy argument is that of the associated actual argument.

While the procedure is active, an allocatable dummy argument array that does not have INTENT(IN) may be allocated, deallocated, defined, or become undefined. Once any of these events have occurred no reference to the associated actual argument via another alias is permitted .

On exit from the routine the actual argument has the allocation status of the allocatable dummy argument (there is no change, of course, if the allocatable dummy argument has INTENT(IN)). The usual rules apply for propagation of the value from the dummy argument to the actual argument.

No automatic deallocation of the allocatable dummy argument occurs as a result of execution of a RETURN or END statement in the procedure of which it is a dummy argument.

Note that an INTENT(IN) allocatable dummy argument array cannot have its allocation status altered within the called procedure. Thus the main difference between such a dummy argument and a normal dummy array is that it might be unallocated on entry (and throughout execution of the procedure).

Example

```
SUBROUTINE LOAD(ARRAY, FILE)
    REAL, ALLOCATABLE, INTENT(OUT) :: ARRAY(:, :, :)
    CHARACTER(LEN=*), INTENT(IN) :: FILE
    INTEGER UNIT, N1, N2, N3
    INTEGER, EXTERNAL :: GET_LUN
    UNIT = GET_LUN()
    OPEN(UNIT, FILE=FILE, FORM='UNFORMATTED')
    READ(UNIT) N1, N2, N3
    ALLOCATE(ARRAY(N1, N2, N3))
    READ(UNIT) ARRAY
    CLOSE(UNIT)
END SUBROUTINE LOAD
```

3.3 Allocatable Array Function Results

An allocatable array function shall have an explicit interface.

On entry to an allocatable array function, the allocation status of the result variable becomes not currently allocated.

The function result variable may be allocated and deallocated any number of times during the execution of the function; however, it shall be currently allocated and have a defined value on exit from the function. Automatic deallocation of the result variable does not occur immediately on exit from the function, but after execution of the statement in which the function reference occurs.[1]

[1] This storage can thus be reclaimed at the same time as that of array temporaries and the results of *explicit-shape-spec* functions referenced in the expression.

Example

```
FUNCTION INQUIRE_FILES_OPEN()
  LOGICAL,ALLOCATABLE :: INQUIRE_FILES_OPEN(:)
  INTEGER I,J
  LOGICAL TEST
  DO I=1000,0,-1
    INQUIRE(UNIT=I,OPENED=TEST,ERR=100)
    IF (TEST) EXIT
100 CONTINUE
  END DO
  ALLOCATE(INQUIRE_FILES_OPEN(0:I))
  DO J=0,I
    INQUIRE(UNIT=J,OPENED=INQUIRE_FILES_OPEN(J))
  END DO
  END
```

3.4 Allocatable Array Components

Allocatable array components are defined to be **ultimate components** just as pointer components are, because the value (if any) is stored separately from the rest of the structure and this storage does not exist (because the array is unallocated) when the structure is created. As with ultimate pointer components, variables containing ultimate allocatable array components are forbidden from appearing directly in input/output lists - the user shall list any allocatable array or pointer component for i/o.

As per allocatable arrays currently, they are forbidden from storage association contexts (so any variable containing an ultimate allocatable array component cannot appear in COMMON or EQUIVALENCE); this maintains the clarity and optimizability of allocatable arrays. However, allocatable array components are permitted in SEQUENCE types, which can be used as global type definitions without recourse to modules.

In a structure constructor for a derived type containing an allocatable array component, the expression corresponding to the allocatable array component must be one of the following:

- an argumentless reference to the intrinsic function `NULL()`; this signifies that the allocatable array component is to have an allocation status of not currently allocated.
- a variable that is itself an allocatable array; the allocatable array component receives the allocation status of the variable, and, if allocated, the shape and value of the variable.
- any other array expression; the allocatable array component has an allocation status of currently allocated with the same shape and value as the expression.

For intrinsic assignment of objects of a derived type containing an allocatable array component, the allocatable array component of the variable on the left-hand-side receives the allocation status and, if allocated, the shape and value of the corresponding component of the expression. This occurs as if the following sequence of steps is carried out:[2]

[2] This ensures that any pointers that point to the previous contents of the allocatable array component of the variable become undefined. Implementations are thus free to skip the allocation-deallocation (or not) when the

1. If the component of the variable is currently allocated, it is deallocated..
2. If the corresponding component of the expression is currently allocated, the component of the variable is allocated with the same shape. The value of the component of the expression is then assigned to the corresponding component of the variable using intrinsic assignment.

component of the variable happens to be allocated with the same shape as the corresponding component of the expression, whichever is most efficient.

Example

```
MODULE REAL_POLYNOMIAL_MODULE
  TYPE REAL_POLYNOMIAL
    REAL, ALLOCATABLE :: COEFF(:)
  END TYPE
  INTERFACE OPERATOR(+)
    MODULE PROCEDURE RP_ADD_RP, RP_ADD_R
  END INTERFACE
CONTAINS
  FUNCTION RP_ADD_R(P1,R)
    TYPE(REAL_POLYNOMIAL) RP_ADD_R, P1
    REAL R
    INTENT(IN) P1,R
    RP_ADD_R%COEFF = P1%COEFF
    RP_ADD_R%COEFF(1) = P1%COEFF(1) + R
  END FUNCTION
  FUNCTION RP_ADD_RP(P1,P2)
    TYPE(REAL_POLYNOMIAL) RP_ADD_RP, P1, P2
    INTENT(IN) P1, P2
    INTEGER M
    ALLOCATE(RP_ADD_RP(MAX(SIZE(P1%COEFF), SIZE(P2%COEFF))))
    M = MIN(SIZE(P1%COEFF, P2%COEFF))
    RP_ADD_RP(:M) = P1%COEFF(:M) + P2%COEFF(:M)
    IF (SIZE(P1%COEFF)>M) THEN
      RP_ADD_RP(M+1:) = P1%COEFF(M+1:)
    ELSE IF (SIZE(P2%COEFF)>M) THEN
      RP_ADD_RP(M+1:) = P2%COEFF(M+1:)
    END IF
  END FUNCTION
END MODULE

PROGRAM EXAMPLE
  USE REAL_POLYNOMIAL_MODULE
  TYPE(REAL_POLYNOMIAL) P, Q, R
  P = REAL_POLYNOMIAL((/4,2,1/))      ! Set P to (X**2+2X+4)
  Q = REAL_POLYNOMIAL((/-1,1/))       ! Set Q to (X-1)
  R = P + Q                           ! Polynomial addition
  PRINT *, 'Coefficients are: ', R%COEFF
END
```

4. Required editorial changes to ISO/IEC 1539-1 : 1996

The following subsections contain the editorial changes to ISO/IEC 1539-1 : 1996 required to include these extensions in a revised definition of the international standard for the Fortran language.
Note, where new syntax rules are inserted they are numbered with a decimal addition to the rule number that precedes them. In the actual document these will have to be properly numbered in the revised sequence.

Comments about each edit to the standard appear within braces {}.

{Page and line number references in these edits are to the March 1996 version of X3J3/96-007.}

4.4, first paragraph, list item (2) [37:42]
Change "nonpointer component that is of derived type,"
To: "component that is of derived type and is not a pointer or allocatable array,"
{The direct component tree stops at allocatable arrays, just as with pointers.}

4.4, second paragraph [38:1]
Insert ", are allocatable arrays" after "of intrinsic type".
{This makes allocatable array components into **ultimate** components, just as pointer components.}

4.4.1, R426 *component-attr-spec* [38:42+]
add new production to rule: "**or** ALLOCATABLE".
{Allow ALLOCATABLE attribute in *component-def-stmt*.}

R427, sixth constraint [39:13]
change "the POINTER attribute is not"
to "neither the POINTER attribute nor the ALLOCATABLE attribute is"
{Do not require an *explicit-shape-spec-list* when ALLOCATABLE is specified.}

Two new constraints at end of list [39:16+]
Add:
"Constraint: If the ALLOCATABLE attribute is specified for a component, the component shall be a deferred-shape array.
Constraint: POINTER and ALLOCATABLE shall not both appear in the same *component-def-stmt*.
{Require ALLOCATABLE components to be deferred-shape arrays. Ensure POINTER and ALLOCATABLE are exclusive.}

R428 *component-initialization* [39:29+]
Add new constraint to end of list:
"Constraint: If the ALLOCATABLE attribute appears in the *component-attr-spec-list*, *component-initialization* shall not appear."
{Forbid default initialization - allocatable array components are already effectively default-initialized to "not currently allocated".}

4.4.1, paragraph beginning "If the **SEQUENCE statement** is" [39:38-39]

add "or allocatable arrays"

after both occurrences of "are not pointers".

{Allocatable array components, like pointer components, stop a SEQUENCE type from being a standard (numeric or character) sequence type.}

4.4.1, after Note 4.25, [42:20+]

add new example:

"Note 4.25.1

A derived type may have a component that is an allocatable array. For example:

```
TYPE STACK
     INTEGER :: INDEX
     INTEGER, ALLOCATABLE :: CONTENTS(:)
END TYPE STACK
```

For each variable of type `STACK`, the shape of component `CONTENTS` is determined by execution of an ALLOCATE statement or evaluation of a structure constructor (of type `STACK`) that is associated with a dummy argument."

{Example needed.}

4.4.4, add new paragraphs to end of section: [45:19+]

"If a component of a derived type is an allocatable array, the corresponding constructor expression shall either be a reference to the intrinsic function `NULL()` with no arguments, an allocatable array, or shall evaluate to an array. If the expression is a reference to the intrinsic function `NULL()`, the corresponding component of the constructor has a status of not currently allocated. If the expression is an allocatable array, the corresponding component of the constructor has the same allocation status as that allocatable array and, if it is allocated, the same shape and value. With any other expression that evaluates to an array the corresponding component of the constructor has an allocation status of currently allocated with the same shape and value as the expression.

Note 4.34.1:

When the constructor is an actual argument, the allocation status of the allocatable array component is available through the associated dummy argument.

If a derived type contains an ultimate allocatable array component, its constructor shall not appear as a *data-stmt-constant* in a DATA statement (5.2.9), as an *initialization-expr* in an *entity-decl* (5.1), or as an *initialization-expr* in a *component-initialization* (4.4.1)."

{Allow structure constructors for derived types with allocatable array components, and define their semantics.}

5.1, eighth constraint, begins "The PARAMETER attribute shall not": [48:12]

After "allocatable arrays,"

Add "derived-type objects with an ultimate component that is an allocatable array or pointer,"

{forbid such objects from having the PARAMETER attribute - unnecessary since it is impossible to construct a value for them as an initialization expression. But that is pretty obscure so it is better to spell it out explicitly.}

5.1, third-last constraint, begins "*initialization* shall not appear": [48:33]
after "an allocatable array,"
add "a derived-type object containing an ultimate allocatable array component,"
{forbid such types from having =*initialization*. This is also unnecessary, but clearer than leaving it to an obscure consequence.}

5.1.2.4.3, second paragraph [55:12]
After "An **allocatable array** is", change "a named array" to "an array".
{Do not insist on allocatable arrays being simple names, i.e. allow components.}

5.1.2.4.3, third paragraph, begins "The ALLOCATABLE attribute may be": [55:15-19]
Replace paragraph with:
"The ALLOCATABLE attribute may be specified for an array in a type declaration statement, a component definition statement, or an ALLOCATABLE statement (5.2.6). An array with the ALLOCATABLE attribute shall be declared with a *deferred-shape-spec-list* in a type declaration statement, an ALLOCATABLE statement, a component definition statement, a DIMENSION statement (5.2.5), or a TARGET statement (5.2.8). The type and type parameters may be specified in a type declaration statement or a component definition statement."

5.2.10, R533-R537 second constraint [61:40]
Change "or an allocatable array"
To "an allocatable array, or a variable of a derived type that has an allocatable array as an ultimate component"
{Forbid initialization of allocatable arrays via the DATA statement.}

5.4, R545 first constraint [66:2-3]
After: ", a pointer,"
Insert "an allocatable array, or"
After "is a pointer"
Delete ",".
{Do not allow derived types containing allocatable arrays in NAMELIST.}

5.5.1, R548 first constraint [66:40]
After "an allocatable array,"
Insert "an object of a derived type containing an allocatable array as an ultimate component,"
{Do not allow derived types containing allocatable arrays in EQUIVALENCE.}

5.5.2, R550 second constraint [69:1]
After "allocatable array,"
Insert "an object of a derived type containing an allocatable array as an ultimate component,"
{Do not allow derived types containing allocatable arrays in COMMON.}

6.1.2, R612-R613, fourth constraint [75:14]
After "shall not have the POINTER attribute"
Insert "or the ALLOCATABLE attribute"
{We do not want to have arrays of allocatable array elements, one from each allocatable array component.}

6.3.1.1, new paragraph at end of section [80:29+]

"If an object of derived type is created by an ALLOCATE statement, any ultimate allocatable components have an allocation status of not currently allocated."

{Specify allocation status of allocatable array components created by an ALLOCATE statement.}

6.3.1.2, new paragraph following the second paragraph [80:42+]

"An allocatable array that is a dummy argument of a procedure receives the allocation status of the actual argument with which it is associated on entry to the procedure. An allocatable array that is an ultimate component of a dummy argument of a procedure receives the allocation status of the corresponding component of the actual argument on entry to the procedure.

{Specify initial status of allocatable dummy arrays. The second sentence is probably unnecessary.}

6.3.1.2, third paragraph [80:43]

After "that is a local variable of a procedure"

Insert "or an ultimate component thereof, that is not a dummy argument (or a subobject thereof)"

{Exclude allocatable dummy arrays from the initial "not currently allocated" status, and also from automatic deallocation.}

6.3.1.2, third paragraph [81:1]

After "If the array"

Add "is not the result variable of the procedure (or a subobject thereof) and"

{Exclude allocatable function results from automatic deallocation.}

6.3.3.1, second paragraph [83:10-13]

After "has the SAVE attribute,"

Add new list items and renumber rest of list:

(2) It is a dummy argument or an ultimate component thereof.

(3) It is a function result variable or an ultimate component thereof.

{Say that these cases retain their allocation status (and thus are excluded from automatic deallocation).}

6.3.3.1, before Note 6.18, [83:18+]

Add new paragraph:

"If a statement contains a reference to a function whose result is an allocatable array or a structure that contains an ultimate allocatable array component, and the function reference is executed, an allocatable array result and any allocated ultimate allocatable array components in the result returned by the function are deallocated after execution of this statement."

{Specify when a function result is deallocated. Perhaps this is not necessary.}

7.1.4.1, fifth paragraph [91:27]

After "returns a disassociated pointer"

Insert "or designates an unallocated allocatable array component of a structure constructor"

After "A disassociated pointer"

Insert "or unallocated allocatable array"

After "with the result" [91:30]

Insert "or by the corresponding component in a structure constructor"

7.1.6.1. [94:6]

After "(3) A structure constructor where each component is an initialization expression"

Insert "and no component has the ALLOCATABLE attribute"

{Exclude structure constructors containing allocatable components from initialization expressions.}

7.5.1.5, paragraph after Note 7.43 [109:35-38]

After "nonpointer components" change "." to

"that are not allocatable arrays. For allocatable array components the following sequence of operations is applied:

1. If the component of *variable* is currently allocated, it is deallocated.
2. If the component of *expr* is currently allocated, the corresponding component of *variable* is allocated with the same shape. The value of the component of *expr* is then assigned to the corresponding component of *variable* using intrinsic assignment."

{Specify semantics to be used for assignment of derived types containing allocatable array components. Note that because pointers to deallocated objects become undefined, this definition does not rule out optimising away the allocation-deallocation when the components are already allocatec with the same shape.}

7.5.1.5, After Note 7.44 [110:5+]

Add new note:

"Note 7.44.1:

If an allocatable array component of *expr* is not currently allocated, the corresponding component of *variable* has an allocation status of not currently allocated after execution of the assignment."

{Note that assignments containing unallocated components are allowed and have the expected effect.}

9.4.2, paragraph after Note 9.26 [149:6]

After "If a derived type ultimately contains a pointer component"

Insert "or an allocatable array component"

{Exclude objects of derived type containing ultimate array components from appearing in i/o statements.}

12.2.1.1 [192:14]

After "whether it is optional (5.1.2.6,5.2.2),"

Insert "whether it is an allocatable array (5.1.2.4.3),"

{ALLOCATABLE-ness of a dummy argument is a characteristic.}

12.2.2 [192:24-25]

After "whether it is a pointer"

Insert "or an allocatable array"

{ALLOCATABLE-ness of a function result is a characteristic.}

After "is not a pointer"

Insert "or an allocatable array"

{shape is not a characteristic for an allocatable array.}

12.3.1.1 item (2) [193:18]

After "assumed-shape array,"

Insert "an allocatable array,"

{Require explicit interface if there is an allocatable dummy array.}

12.4.1.1 [201:21+]

After the paragraph beginning "If a dummy argument is an assumed-shape array"

Add a new paragraph:

"If a dummy argument is an allocatable array the actual argument shall be an allocatable array and the types, type parameters and ranks shall agree. It is permissible for the actual argument to have an allocation status of not currently allocated."

{Requirements for arguments associated with an allocatable dummy array.}

12.4.1.6, item (1) of first paragraph [203:28-29]

Replace "No action that affects the allocation status may be taken."

With "Action that affects the allocation status of the entity or any part thereof shall be taken through the dummy argument."

{Allow ALLOCATE/DEALLOCATE via the dummy whilst prohibiting it via any other alias.}

12.4.1.6, item (2) of first paragraph [205:5]

After "If the pointer association status"

Insert "or the allocation status"

{After ALLOCATE/DEALLOCATE of the dummy, prohibit all other accesses to the actual argument.}

13.14.79,

After "a disassociated pointer" [259:26]

Insert "or unallocated allocatable array"

After "disassociated association status" [259:33]

Insert "or, when corresponding to an allocatable array component in a structure constructor, an unallocated allocatable array"

Annex A, entry "**allocatable array**" [293:12-13]

Change "A named array"

To "An array"

Add new sentence to end of entry "An allocatable array may be a named array or a *structure component*."

Annex A, entry "**direct component**" [295:38]

Change "nonpointer component that is of derived type"

To "component that is of derived type and is not a pointer or allocatable array,"

Annex A, entry "**ultimate component**" [301:11-13]

After "is of *intrinsic type*"

Insert ", has the ALLOCATABLE attribute,"

After "does not have the POINTER attribute"

Insert "or the ALLOCATABLE attribute"