ISO/IEC JTC1/SC22/WG5 N1173

International Standards Organization

Data Type Enhancements

in

Fortran

Technical Report defining extension to
ISO/IEC 1539-1 : 1996

{Produced 18-Dec-95}

THIS PAGE TO BE REPLACED BY ISO CS

Contents

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 incorportated 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 enhancements defined in this technical report cover two main areas. The first extends the capability of parameterization defined for intrinsic types to derived types and the second 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 (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.

* * *

Parameterized derived types are required for two main reasons. Firstly, there are many circumstances where a derived type is required to work together with intrinsic types where the ability to parameterize the kind of the latter and not the former causes very considerable problems. In one case different versions of the program can be selected by the use of the parameter but to enable the derived type to properly interwork a different type with a different name must be used. This results in very clumsy and inflexible programs and a significant program maintenance overhead, substantially defeating the object of the kind parameterization. Secondly, there are a large number of types where there is a need to manipulate objects where the only difference between various entities is in the size of some internal component. For example, there are entities like vectors that may differ in the dimensionality of the space they span and therefore in the number of reals that are involved in their representation, or in matrices that differ in their order. These are very like the intrinsic character data type where data objects may differ in the number of characters in the string and where this is specified by a length parameter on the type. This is clearly preferable to having multiple separate types which differ only in such a size determining property. Both these requirements are met by the addition of parameterized derived types to the language.

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 and for user defined parameterized derived types.

3.1 ALLOCATABLE Attribute Extension

3.1.1 Allocatable Attribute Regularisation

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.

3.1.2 Allocatable Arrays as Dummy Arguments

An allocatable dummy argument array must 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 since an INTENT(IN) allocatable dummy argument array cannot have its allocation status altered, it is not seem to be of very much use (the only difference between such a dummy argument and a normal dummy array is that it might be, and thus remain, unallocated).

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, FORMAT='UNFORMATTED')
    READ(UNIT) N1, N2, N3
    ALLOCATE(ARRAY(N1, N2, N3))
    REAL(UNIT) ARRAY
    CLOSE(UNIT)
END SUBROUTINE LOAD
```

Implementation Cost

Minimal, similar to pointer dummy arrays (except that the descriptor need not be so big).

3.1.3 Allocatable Array Function Results

An allocatable array function must have an explicit interface.

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

The result of the function must be allocated and defined at the time of exit from the function. No automatic deallocation of the result variable occurs on exit from the function; however, this does occur after execution of the statement in which the function reference occurs.[1]

Example

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

Implementation Cost

Minimal, mostly just to establish an appropriate calling convention. Deallocation of the result can be handled exactly as *explicit-shape-spec* array functions currently.

3.1.4 Allocatable Array Components

Allocatable array components are ultimate components because the value is stored somewhere else (and they do not come into existance with the rest of the structure, just like pointers). As with ultimate pointer components, variables containing ultimate allocatable array components are forbidden from appearing directly in input/output lists - the user must 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.

Example

{To be added}

Implementation Cost

---

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

Minimal. It will be necessary to store a descriptor in the derived type, but this need not be as powerful as a pointer array descriptor.

3.2 Description of parameterized derived type enhancements

There are seven main areas of language design where an extension such as this impacts the existing language and where syntax and semantics must be defined. These are:

- the definition of the type,
- declaration of objects of such a type,
- constructing a value of such a type,
- inquiring as to the value of a type parameter for an existing object of such a type,
- intrinsic assignment for objects of such a type,
- argument association and overload resolution, and
- the visibility and scoping rules.

Syntactic forms and semantic rules exist covering the use of parameterized intrinsic types in all but the first of these areas; for obvious reasons there is no type definition for an intrinsic type. The aim of all of the following is to extend the notion of type parameterization to user defined types in such a way as to make the manipulation of derived types and intrinsic types as consistent and as similar as possible.

In this section the technical nature of the proposal in each of the above areas is covered with sufficient detail to indicate the essential nature of the proposed syntax and semantics. This is done informally with the approach illustrated by example rather than with detailed syntactic and semantic rules. These formal rules will be defined in subsequent sections in the form of proposed edits to the current international standard for the programming language Fortran which would implement the proposed extensions.

All parameters for intrinsic types are quantities of type default integer. This technical report proposes that parameters for derived types be similarly restricted at this time; however, the detailed form of the extension defined in this technical report is such that parameters of other types could be added by further extension if that proves to be desirable.

The intrinsic types have parameters of two quite different natures. There are the static parameters that determine the nature of the machine representation. These are all characterised for the intrinsic types by the same parameter name, **`KIND`**. This is used both for the keyword in the *type-spec* and as the generic name of the parameter-value inquiry function for such a parameter. The other parameter variety, where the value is not necessarily static, only applies intrinsically for the character type. Here the parameter, **`LEN`**, determines the length or the number of characters in the datum. As for **`KIND`**, the name **`LEN`** is also both the parameter keyword name and the generic name of the inquiry function used to find the value of the parameter for an appropriate data object.

This technical report defines the extension to derived types of parameterization in such a way as to allow for any number of both sorts of type parameter. It also preserves the consistency rule that the keyword name for a type parameter is also the generic name of an inquiry function that may be used for inquiring as to the actual type parameter values for any given object of a parameterized type. This provides for full regularity of treatment between intrinsic and derived types.

3.2.1 The Type Definition

The syntax optionally allows a list of dummy type parameter names to be added in parentheses following the type-name in the type-definition statement. These dummy type parameters are permitted as primaries in the expressions used to specify the attributes of the various components of the type. The inclusion of names in such a list of dummy type parameter names is considered to declare these names to be type parameter names and to be of default integer type. It is assumed that the majority of parameters will be "size-determining", like **LEN**, and so the default for a parameter that is declared only on the type-definition statement is that it is not "kind-determining". Type parameters that are to be used to determine the kind of a component must be distinguished by being declared as such. This is to be done by declaring all such type parameters to have the **KIND** attribute within the body of the type definition.

For example, the extended syntax would allow a type definition such as,

```
TYPE MATRIX(wkp,dim)
  KIND :: wkp
  REAL(wkp),DIMENSION(dim,dim) :: element
ENDTYPE MATRIX
```

Where **wkp** and **dim** are dummy type parameters; one used to determine the kind of the matrix elements and the other the order of the matrix.

The expressions used to declare the attributes of components of a parameterized type shall be constant expressions, possibly involving the type parameters as primaries. Any parameter used to determine the kind of a component must be declared as a static "kind" parameter and any type parameter not declared in this way with the kind attribute shall not be used to determine the kind of components.

Parameterized types may be declared to have the **SEQUENCE** property. Two sequence types are the same if and only if they have the same name, define the same type parameters of the same kind in the same order and define the same components with the same names and the same dependencies on the type parameters. Two objects of a parameterized sequence type can become storage associated only when their sequence types are the same and they have the same parameters with the same values. Even if all components of such a type are numeric sequence types, a parameterized sequence type shall not be considered to be a numeric sequence type.

3.2.2 Object declaration

Objects of a parameterized type shall be declared in ways entirely analogous to those used for intrinsic types. Where the type has parameters, actual values shall be provided for these parameters when objects of such types are declared. For example, objects of the above matrix type could be declared,

```
type(MATRIX(4,3)) :: rotate,trans
type(MATRIX(KIND(0.0),4)) :: metric
type(MATRIX(wkp=8,dim=35)) :: weight
type(MATRIX(wkp=8,dim=*)) :: hessian
type(MATRIX(dim=2*n+1,wkp=4)) :: distance
```

For purposes of illustration it could be considered that the last two declarations are in subprogram units where the value, **n**, is possibly that of a dummy variable. In such a context the last statement would be declaring an automatic or dummy object and the next to last would be declaring a dummy argument that was to assume the value for the the **dim** type parameter from that of the associated actual argument, c.f. similar usage with the length parameter for characters.

Where a type is defined with parameters, the *type-spec* in an object declaration shall specify actual values to be used to supply values for the dummy type parameters that determine the attributes of the components as defined in the type definition. These actual type parameter values shall be specified as if they were an actual argument list following the type name. The association between actual and dummy type parameters may be positional or keyword and the same rules as for argument association shall apply.

Any value that becomes associated with a type parameter declared to have the kind attribute shall be specified by an integer scalar initialization expression. The rules for type parameters without the kind attribute are the same as for the intrinsic **LEN** type parameter; in particular the actual values for such parameters shall be specification expressions or assumed. If such an object is to appear in a common or equivalence context the type must have the sequence property and the actual type parameter values must be constant.

{{{ Note, at this time the possibility of derived types having parameters with the ***OPTIONAL*** *attribute is not being proposed. However, such a future extension is not ruled out. It would be possible for a parameter to be declared as optional in the type definition, and some suitable syntax for providing a default value to be used when an actual value is omitted. This added capability is cosmetic and although possibly desirable is considered to add unnecessary complexity at this stage. }}}*

3.2.3 The form of the Constructor

The syntactic similarity of a type value-constructor and a generic function reference is recognised and extended. The constructor reference is defined to be identical to a function reference. The model of the **REAL** type conversion function is used and extended to all derived types. The constructor is therefore defined to be an intrinsic procedure with generic name the same as the type name. Where the type is parameterized the constructor reference shall include the parameters as an extra set of arguments following the list of component expressions. The analogy with

```
REAL(A,KIND)
```

for the matrix type would be

```
MATRIX(element,wkp,dim)
```

where the expression associated with the element component would need to be assignment conformant with a rank 2 array of shape **(/dim,dim/)** and type real of **KIND=wkp**.

The general form is therefore,

type-name(*component-expr-list*,*type-param-expr-list*)

which is identical to the function reference syntax,

function-name(argument-expr-list)

provided the keyword names that correspond to the argument keywords for the component expressions are the component names and for the parameter expressions the keyword names are those of the type parameterers. A constructor reference of the following form now becomes valid,

```
MATRIX(wkp=4,dim=10,element=0.0)
```

As a further example of the greater expressive utility provided consider the constructors produced by the following. Given a type defined by,

```
TYPE STOCK_ITEM
  INTEGER :: id,holding,buy_level
  CHARACTER(LEN=20) :: desc
  REAL :: buy_price,sell_price
ENDTYPE STOCK_ITEM
```

the two constructor references below would mean the same thing.

```
STOCK_ITEM(12345,75,10,"Pencils HB",1.56,2.49)

STOCK_ITEM(desc="Pencils HB", id=12345, &
           holding=75, sell_price=2.49, &
           buy_level=10, buy_price=1.56 )
```

By defining a constructer reference to be a function reference, the assignment semantics that apply the correspondence of component to expression, in effect means that the constructor name is generic. The set of overloads are defined as all those which would produce valid assignments to each of the components. For pointer components the keyword is the component name and the semantics that apply to the expression to component correspondence is that of pointer assignment; the expression in this case shall deliver a result that has the target attribute. This provides for the constructor exactly the same relationship between actual expression and corresponding components as between actual and dummy argument of the relevant characteristics.

As with function reference actual arguments, positional correspondence shall be permitted up to the first use of the keyword form, all subsequent component/expression arguments would have to be of the keyword form.

3.2.4 Type parameter value inquiry

For each type parameter declared as part of a type definition there is a generic function with the type parameter name as its generic name. This function takes as its single non-optional argument any entity of any rank of the relevant type and it delivers an integer valued result which is the value of the named type parameter.

If the parameter inquired about is a kind parameter the inquiry function may appear in initialization expressions. If the parameter is non-kind parameter, the inquiry function may appear in specification expressions. In both these cases the same restrictions that apply to the **KIND** and **LEN** functions also apply.

In general such type parameter inquiry functions have the same scope and accessibility as the type to which they relate. However, if the type is defined in a module the visibility of the type name and the type parameter names can be controlled separately, although it would be unusual for such separate control to be exercised. The rule to be applied is that if a type parameter name is declared to be private to a module, neither the inquiry function of that name nor the type parameter keyword are visible in a using program. Similarly if access to a type parameter name is denied because of a **USE** statement with an **ONLY** clause, neither the inquiry function nor the keyword are accessible. All declarations of objects of the associated type would have to use the positional form of actual parameter specification. Finally if such a name is renamed on a USE statement both the function and the type parameter uses are locally renamed.

3.2.5 Intrinsic assignment

Intrinsic assignment is to be defined only when the variable and expression have the same type and type parameter values.

3.2.6 Argument association and overload rules

The rules for argument association shall be that dummy argument and actual argument must match in type and type parameters, as for objects of intrinsic types. This matching of parameters may be achieved for nonkind parameters by the dummy argument assuming its type parameter values from the associated actual argument. The kind type parameters cannot be assumed; they must always be explicitly and statically specified, but as with intrinsic kind parameters these may be used to resolve generic overloads.

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 {}.

4.1 Edits to implement ALLOCATABLE attribute extension

{Page and line number references in these edits are to the F95CD.}

4.4, first paragraph [37/39]
insert ", are allocatable arrays" before "or are pointers".
{This makes allocatable array components into **ultimate** components, just as pointer components.}

4.4.1, R426 *component-attr-spec* [38:31+]
add new production to rule: "**or** ALLOCATABLE".
{Allow ALLOCATABLE attribute in *component-def-stmt*.}

R427, sixth constraint [38:45]
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 [39:1+]
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* [39:3+]
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" [39:22-23]
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, [41:45+]

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: [45:15+]

"If a component of a derived type is an allocatable array, the corresponding constructor expression shall evaluate to an array. The value of the constructor will have a component that has an allocation status of currently allocated with contents given by the constructor expression.

Note 4.34.1:

The allocation status of the allocatable array component is available to the user program only if the constructor is associated with a dummy argument. Also, when the constructor value is used in an assignment, the corresponding component of the variable being defined shall already be allocated with the same shape as the component in the constructor.

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.}

5.1, third-last constraint, begins "*initialization* shall not appear": [48:29]

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.}

5.1.2.4.3, second paragraph [54:41]

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": [55:1-5]

Replace paragraph with:

"The ALLOCATABLE attribute may be specified 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."

5.2.10, R533-R537 second constraint [61:10]

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 [65:19]

Change "is a pointer"

To "is a pointer or allocatable array"

{Do not allow derived types containing allocatable arrays in NAMELIST.}

5.5.1, R548 first constraint [66:13]

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 [68:18]

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 [73:14]

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 [78:26+]

"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 [78:38+]

"An allocatable array that is a dummy argument of a procedure receives the allocation status of the actual argument with which it is associatedon 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 [78:39]

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 [78:42]

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 [80:36-39]

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 6.17, [81:2+]

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.}

7.1.6.1. [92:7]

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, before Note 7.46 [108:34+]

Add new note:

"Note 7.45.1: For an ultimate component that is an allocatable array, the component in the variable being defined must already be allocated, and its shape must be the same as that of the corresponding component of the expression.."

{Specify semantics to be used for assignment of derived types containing allocatable array components.}

{Note that this is quite a hardship, effectively requiring the user to overload assignment if he wants it to be useful. There are no real reasons why these semantics should be required other than that of minimal change to the document, minimal extension to the language, and minimal effort for the implementors. Alternative semantics for this construct are described elsewhere.}

9.4.2, paragraph after Note 9.26 [147:6]

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 [190:14]
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 [190:25]
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) [191:19]
After "assumed-shape array,"
Insert "an allocatable array,"
{Require explicit interface if there is an allocatable dummy array.}

12.4.1.1 [200:27+]
Add new paragraph to end of section before Note 12.18
"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 current allocated."
{Requirements for arguments associated with an allocatable dummy array.}

12.4.1.6, item (1) of first paragraph [201:44]
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 [203:29]
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.}

Annex A, entry "**allocatable array**" [289:12-13]
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 "**ultimate component**" [295:5-7]
After "is of *intrinsic type*"

Insert ", has the ALLOCATABLE attribute,"
After "does not have the POINTER attribute"
Add "or the ALLOCATABLE attribute"

4.2Edits to implement parameterized derived types

2.4.1.2 [15/19]
before "agreement" add "and type parameter"

4 [29/15]
replace "Intrinsic data-types are" by "Data types may be"

{{{NOTE: FROM HERE ON REFERENCES ARE TO F90 NOT F95CD}}}

4.3.1.1 [27/8],
4.3.1.2 [28/25],
4.3.2.1 [30/21],
4.3.2.2 [32/11]
After "(13.13.51)" add "or by a type parameter value selector (4.4.1)"

4.3.1.3 [29/30]
Add sentence "The kind type parameter of an approximation method used for the parts of a complex value are returned by the intrinsic inquiry function KIND (13.13.51) or by a type parameter value selector (4.4.1)."

4.4.1 [32/41]
Add line following
[*param-spec-stmt*]...

4.4.1 [33/3]
Add to end of R424 "[*(dummy-type-param-list)*]"

Add new rule
R424.1 *dummy-type-param* **is** *type-param-name*

4.4.1 [33/16]
Add following
R425.1 *param-spec-stmt* **is** KIND [::] *dummy-type-param-list*

A dummy type parameter that is specified in a *param-spec-stmt* with a KIND attribute is a **kind type parameter**. Any other dummy type parameter is a **nonkind type parameter**, and must not be used to determine the values of actual kind type parameters of components.

4.4.1 [33/26]
Add constraints
Constraint: If the *type-spec* specifies a value for a kind type parameter, this must be a scalar integer initialization expression, possibly involving as primaries the names of one or more dummy kind type parameters specified on the *derived-type-stmt*.

Constraint: If the *type-spec* specifies a value for a nonkind type parameter, this must be a scalar integer constant expression, possibly involving as primaries dummy type parameter names specified on the *derived-type-stmt*.

4.4.1 [33/36 & 38]

After ")" add ", possibly involving as primaries dummy type parameter names specified on the *derived-type-stmt*."

4.4.1 [33/38+]

Add following paragraph

If the type has type parameters, actual values for these must be specified when an entity of this type is declared or constructed. These values may be used via the associated dummy type parameter names to specify array bounds and type parameter values for components of the type.

4.4.1 [34/34]

Add following paragraph

Examples of type definitions with type parameters are:

```
TYPE VECTOR(WP, ORDER)
  KIND :: WP
  REAL(KIND=WP) :: comp(1:ORDER)
ENDTYPE VECTOR
```

Objects of type VECTOR could be declared:

```
TYPE(VECTOR(WP=KIND(0.0),ORDER=3)) :: rotation
TYPE(VECTOR(WP=KIND(0.0D0),ORDER=100)) :: steepest
```

The scalar variable **`rotation`** is a three-vector with each component represented by a default real. The scalar vector **`steepest`** is vector in a 100 dimension space and each component is represented by a double precision real.

4.2.1 Edits to implement inquiry function

4.4.1 [34/34]

continue adding the following text

For each type parameter specified there is a generic inquiry function that has the same name as the type parameter. This function takes as its single nonoptional argument any entity of the derived type, and it returns as its result the integer value for this named type parameter that applies for its argument. For example, **`WP(rotation)`** would return 4 on a system where 4 was the default real kind and **`ORDER(steepest)`**would return 100. Note, the argument of such a type parameter inquiry function may be of any rank.

4.4.4 [37/3]

Replace "value of" by "value of the"

4.4.4 [37/5]

Replace "*expr-list*" by "*expr-list*[,*type-param-expr-list*]"

Add constraint

Constraint: If the derived type has one or more type parameters, the *type-param-expr-list* must be present with the same number of expressions. If the derived type has no parameters, the *type-param-expr-list* must not be present.

Constraint: If the derived type has one or more kind type parameters, each corresponding *type-param-expr* must be an initialization expression.

4.4.4 [37/10]

Before "A structure" add

The type parameter expressions, if present, provide values for the type parameters of the type and hence control the shapes and type parameters of the components.

4.4.4 [37/16]

Add the following paragraph

An example of a constructor for a parameterized type is:

```
VECTOR(0.0,KIND(0.0D0),3)
```

This would construct a three-vector whose components were all zero and of double precision.

4.4.4 [37/5]

Replace "*expr-list*" with "*comp-expr-list*"

Add

R430.1 *comp-expr* **is** [*component-name*=]*expr*

Constraint: Each *component-name* must be the name of a component specified in the type definition for the type-name.

Constraint: The *component-name*= may be omitted only if it has been omitted from each preceding *comp-expr* in the *comp-expr-list*.

4.4.4 [37/7]

After "type." add sentence

The correspondence between expression and component may be indicated by the component name appearing explicitly in the form of a keyword in a manner similar to procedure argument association (12.4.1).

4.2.2 Constructor as generic function

4.4.4 [37/2]

After "corresponding" add "generic function reference that is a"

5.1 [39/24]

Replace "*type-name*" by "*type-name*[*type-selector*]"

5.1 [39/39]
Add constraint
Constraint: The *type-selector* must appear if the type is parameterized and must not appear otherwise.

5.1.1.7 [43/23]
Add rules and constraints

R509.1 *type-selector* is (*type-param-selector-list*)

R509.2 *type-param-selector* is [*type-param-name*=]*type-param-expr*

R509.3 *type-param-expr* is *scalar-int-initialization-expr*
or *type-param-value*

Constraint: There must be one and only one *type-param-selector* corresponding to each type parameter of the type.

Constraint: The *type-param-expr* must be a *scalar-int-initialization-expr* if the corresponding type parameter is a kind type parameter.

Constraint: The *type-param-name*= may be omitted if it was omitted from all previous *type-param-selector* in the list.

The type selector, if present, specifies values for the type parameters of the type and hence the type parameters and shapes of the components of the type.

{{{{Note for the editor: The rules associated with type-param-values, in particular automatic and assumed type parameters, appear to be covered in the description for character length. A reordering of this material might read better but I think the current text is actually correct even though it now has extended effect. It is probably now in the wrong place in the chapter.}}}}

5.5.2.3 [59/37]
After "type" add "and type parameters"

7.1.4.2 [76/24]
After the second "The type" add "and type parameters."

7.1.6.1 [77/25 & 78/7]
After "KIND" add
", a derived-type kind type parameter inquiry function"

7.1.6.2 [79/12]
After "KIND" add
", a derived-type type parameter inquiry function"

7.1.7 [80/29]
Add paragraph

The appearance of a structure constructor requires the evaluation of the component expressions and may require the evaluation of type parameter expressions. The type of an expression in which a structure constructor appears does not affect, and is not effected by, the evaluation of such expressions, except that evaluation of the kind type parameters may affect the resolution of a generic reference to a defined operation or function and hence may affect the expression type.

7.5.1.2 [89/28]

Replace "type," by "type and the same type parameter values,"

7.5.1.2 [89/41]

Replace "type as" by "type and the same type parameter values as"

11.3.2 [158/16]

Add sentence

If a derived type type parameter is renamed, the local name is used for both the type parameter inquiry function and the type parameter keyword name used when specifying actual type parameter values.

[158/32+]

Add paragraph

Note that, if a type-name is inaccessible, the type parameter inquiry names, if any, for the type may still be accessible. However, such an inquiry function can only be invoked with an argument that is also accessed by use association. If a type parameter name is inaccessible but the type is accessible, objects of this type must be declared using the positional specification of the relevant actual parameter and no reference may be made to the corresponding inquiry function by this name.

12.2.1.1 [166/6]

Replace "or character length" by " character length, or nonkind type parameter"

12.3.1.1 [167/2]

Replace "that" by "that assumes the value for a nonkind derived type parameter or that"

12.3.1.1 [167/4]

Add additional item to list and renumber list

(e) A result with a nonconstant type parameter value (derived type functions only)

12.4.1.1 [172/41]

Add sentence

The value of a type parameter of an actual argument of a derived type must agree with the corresponding value for the dummy argument.

14.1.2 [241/26]

Replace ", in" by " and type parameters, in"