04/07/96 13:59

International Standards Organisation

Parameterized Derived Types

in

Fortran

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

{Produced 4-Jul-96}

THIS PAGE TO BE REPLACED BY ISO CS

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: Parameterized Derived Types

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 extends the capability of parameterization defined for intrinsic types to derived types.

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

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 for user defined parameterized derived types.

3.1 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.

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. The 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. **`KIND`** parameters can be used to resolve overloads.

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 extends parameterization to derived types in such a way as to allow for any number of both sorts of type parameter. It also introduces more general **static** parameters for derived types which can be used to resolve generic overloads but which are not necessarily **`KIND`** type parameters. Type parameters that are not static are called **nonstatic** type parameters.

<<<<<<<<<<<<<<<<<<<< **Start of Text option 1**>>>>>>>>>>>>>>>>>>>>
Parameters are treated in a similar way to the components of a derived type. However this is not necessarily meant to imply a similar implementation model. When viewed as components, the main difference between parameters and components is that parameter ‘components’ are defined to be ‘read-only’ i.e. they cannot be accessed in such a way as to change their value.
<<<<<<<<<<<<<<<<<<<<**End of Text option 1**>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<**Start of Text option 2** >>>>>>>>>>>>>>>>>>>>
This technical report extends parameterization to derived types 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.
<<<<<<<<<<<<<<<<<<<<**End of Text option 2**>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<**Start of Text option 1** >>>>>>>>>>>>>>>>>>>>

3.1.1 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 type parameters must be specified in the body of the type definition in the same way as for components. Type parameters, however, can only be specified as default **`INTEGER`**.

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

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

Where **`wkp`** and **`dim`** are dummy type parameters. In this example the nature of the parameters is determined implicitly. Thus since **`wkp`** is used to determine the **`KIND`** of **`element`** it is implicitly static and default integer whilst since **`dim`** is not used to determine the **`KIND`** of a component it is implicitly declared to be nonstatic and default integer.

The above example could also be written as :-

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

where **`wkp`** has now been declared explicitly to be static.

As indicated previously, the declaration of **wkp** as **STATIC** in the above example is not necessary since **wkp** is used to determine the kind of a component and hence must be static. The ability to declare static parameters explicitly is provided for cases where it is required to use a parameter (which is not used to determine the kind of a component) to resolve generic function overloads.

For example in,

```
TYPE T(d)
  INTEGER, STATIC :: d
  REAL :: element(d)
ENDTYPE
```

d is declared explicitly to be static. Therefore even though it is not used to specify the kind of a component it can still be used to resolve generic function overloads.

<<<<<<<<<<<<<<<<<<<<End of Text option 1 >>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<Start of Text option 2 >>>>>>>>>>>>>>>>>>>>

3.1.2 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.

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

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

Where **wkp** and **dim** are dummy type parameters. In this example the nature of the parameters is determined implicitly since there is no declaration. Thus since **wkp** is used to determine the **KIND** of **element** it is implicitly static and default integer whilst since **dim** is not used to determine the **KIND** of a component it is implicitly declared to be nonstatic and default integer.

The above example could therefore also be written as :-

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

where **wkp** and **dim** have now been declared explicitly.

As mentioned previously, the declarations of **wkp** and **dim** in the above example are redundant since **wkp** is used to determine the kind of a component and hence must be static and since **dim** is not used to determine the kind of a component it must be nonstatic. The ability to declare parameters explicitly is provided for cases where it is required to use a parameter (which is not used to determine the kind of a component) to resolve generic function overloads. It is also provided to allow for possible later extension to allow real parameters.

It follows from the above that a parameter which is to be used to resolve generic overloads, but which is not subsequently used to determine the **`KIND`** of a component in the derived type definition, must be declared explicitly with the **`STATIC`** attribute. For example in,

```
TYPE T(d)
  INTEGER, STATIC :: d
  REAL :: element(d)
ENDTYPE
```

d is declared explicitly to be static. Therefore even though it is not used to specify the kind of a component it can still be used to resolve generic function overloads.

<<<<<<<<<<<<<<<<<<<<**End of Text option 2** >>>>>>>>>>>>>>>>>>>>

Parameterized derived 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.

3.1.3 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 (implicitly or explicitly) to have the **`STATIC`** attribute shall be specified by an integer scalar initialization expression. The rules for type parameters without the **`STATIC`** 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. }}}*

3.1.4 The form of the Constructor

<<<<<<<<<<<<<<<<<<<<**Start of Text option 1** >>>>>>>>>>>>>>>>>>>>

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 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 preceding the list of component expressions. This is consistent with the intuitive ordering of the parameters in the component definitions.

For the matrix type we would have, for example,

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

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`**.

<<<<<<<<<<<<<<<<<<<<**End of Text option 1** >>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<**Start of Text option 2** >>>>>>>>>>>>>>>>>>>>

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`.

<<<<<<<<<<<<<<<<<<<<**End of Text option 2** >>>>>>>>>>>>>>>>>>>>

The general form is therefore,

<<<<<<<<<<<<<<<<<<<<**Start of Text option 1** >>>>>>>>>>>>>>>>>>>>

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

<<<<<<<<<<<<<<<<<<<<**End of Text option 1** >>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<**Start of Text option 2** >>>>>>>>>>>>>>>>>>>>

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

<<<<<<<<<<<<<<<<<<<<**End of Text option 2** >>>>>>>>>>>>>>>>>>>>

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 parameters. A constructor reference of the following form now becomes valid,

<<<<<<<<<<<<<<<<<<<<**Start of Text option 1** >>>>>>>>>>>>>>>>>>>>

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

<<<<<<<<<<<<<<<<<<<<**End of Text option 1** >>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<**Start of Text option2** >>>>>>>>>>>>>>>>>>>>

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

<<<<<<<<<<<<<<<<<<<<**End of Text option2** >>>>>>>>>>>>>>>>>>>>

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 constructor reference to be a function reference, the assignment semantics that apply to 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 arguments 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.

<<<<<<<<<<<<<<<<<<<<Start of Text option 1>>>>>>>>>>>>>>>>>>>>

3.1.5 Type parameter value inquiry

Inquiry for parameter values is via the component selection syntax. For example, the values of the **`wkp`** and **`dim`** parameters for a matrix object M (defined using the definition above) could be obtained via the expressions, **`M%wkp`** and **`M%dim`**.

The restriction on parameter components being read-only means that statements like,

```
M%wkp = 8
M%dim = 5
READ(*,*) M%wkp
```

are all illegal.

If the parameter inquired about is a static parameter the inquiry expression may appear in initialization expressions. If the parameter is a non-static parameter, the inquiry expression may appear in specification expressions.

The component selection form of inquiry is extended to the intrinsic types. Thus parameter values may be obtained via **`object%KIND`** for **`INTEGER`**, **`REAL`** , or **`CHARACTER objects`** and **`object%LEN`** for character objects.

<<<<<<<<<<<<<<<<<<<<End of Text option 1>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<Start of Text option 2>>>>>>>>>>>>>>>>>>>>

3.1.4 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.

<<<<<<<<<<<<<<<<<<<<End of Text option 2>>>>>>>>>>>>>>>>>>>>

3.1.6 Intrinsic assignment

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

3.1.7Argument 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 non-static parameters by the dummy argument assuming its type parameter values from the associated actual argument. The static type parameters cannot be assumed. They must always be explicitly and statically specified. As with intrinsic kind parameters, static type parameters may be used to resolve generic overloads.

3.1.8Visibility and Scoping rules

As far as type parameters are concerned the visibility and scoping rules are determined in the same way as for ordinary components of derived types.

Thus parameters declared to be private cannot have their values inquired on and cannot be referenced by keyword in declarations or in constructors.

A USE with a rename is deemed to rename both the keyword and the parameter name.

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 {}.

N.B. In this draft the edits refer to X3J3/96-007R1, April 22, 1996, 8:35 a.m.

4.1.1Edits to implement parameterized derived types

2.4.1.2 [15/24] before “The only” add sentence “Derived types may be parameterized, with user-defined parameters, in a similar way to the intrinsic types.”

[15/25]

before "agreement" add "and type parameter"

4 [29/16]

replace "Intrinsic data-types are" by "Data types may be"

[29/25]

replace “type” by “type, its type parameters (if any),”

[29/26]

replace “components” by “components. Type parameters are defined implicitly or explicitly (4.4.1).”

4.3.1.1 [31/17],
4.3.1.2 [32/39],
4.3.1.3 [34/26],
4.3.2.1 [35/18],
4.3.2.2 [37/22]

After "(13.14.52)" add "or by a type parameter value inquiry(4.5)"

4.4 [37/39]

Replace “type” by “, the name of its parameters, if any,”

<<<<<<<<<<<<<<<<<<<<**Start of Text Option 1** >>>>>>>>>>>>>>>>>>>>

BLANK

<<<<<<<<<<<<<<<<<<<<**End of Text Option 1** >>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<< **Start of Text Option 2** >>>>>>>>>>>>>>>>>>>>

4.4.1 [38/25]

Add line following

[*param-def-stmt*]...

<<<<<<<<<<<<<<<<<<<< **End of Text Option 2** >>>>>>>>>>>>>>>>>>>>

4.4.1 [38/29]

Add to end of R423 "[*(dummy-type-param-list)*]"

Add new rule

R424.1 *dummy-type-param* **is** *type-param-name*

<<<<<<<<<<<<<<<<<<<< **Start of Text Option 1** >>>>>>>>>>>>>>>>>>>>

4.4.1 [38/42+]

Add line

or **STATIC**

Constraint: If a component is a dummy type parameter then the *component-attr-spec* may not be **`POINTER`** or **`DIMENSION`** and the *type-spec* (R425) must specify default integer (R502, 5.1.1.1)

<<<<<<<<<<<<<<<<<<<< **End of Text Option 1** >>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<< **Start of Text Option 2** >>>>>>>>>>>>>>>>>>>>

4.4.1 [38/39]

Add following

R424.2 *param-def-stmt* **is** **`INTEGER [,STATIC ::]`** *type-param-list*

*{ N.B. The alternative of spelling "**`INTEGER, [STATIC]`**" as just "**`STATIC`**" or any other spelling is a trivial alteration***}**

<<<<<<<<<<<<<<<<<<<< **End of Text Option 2** >>>>>>>>>>>>>>>>>>>>

Add following text:

<<<<<<<<<<<<<<<<<<<< **Start of Text Option 1** >>>>>>>>>>>>>>>>>>>>

A *dummy-type-param* that is specified with a **`STATIC`** attribute is a **static type parameter**. A *dummy-type-param* that is specified without the **`STATIC`** attribute is a **nonstatic type parameter**, and must not be used to determine the values of actual kind type parameters of components.

Dummy type parameters that are specified in a *component-def-stmt* without the **`STATIC`** attribute are static if they are subsequently used to determine the kind of a component directly or indirectly, otherwise they are nonstatic.

<<<<<<<<<<<<<<<<<<<< **End of Text Option 1** >>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<< **Start of Text Option 2** >>>>>>>>>>>>>>>>>>>>

A *dummy-type-param* that is specified in a *param-def-stmt* with a **`STATIC`** attribute is a **static type parameter**. A *dummy-type-param* that is specified in a *param-def-stmt* without the **`STATIC`** attribute is a **nonstatic type parameter**, and must not be used to determine the values of actual kind type parameters of components.

Dummy type parameters that are not specified in a *param-def-stmt* are static if they are subsequently used to determine the kind of a component directly or indirectly, otherwise they are nonstatic.

<<<<<<<<<<<<<<<<<<<< **End of Text Option 2** >>>>>>>>>>>>>>>>>>>>

Add the following text:

4.4.1 [39/16+]

Add constraints

Constraint: If the *type-spec* specifies a value for a static type parameter, this must be a scalar integer initialization expression, possibly involving as primaries the names of one or more dummy static type parameters specified on the *derived-type-stmt*.

Constraint: If the *type-spec* specifies a value for a nonstatic 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 [39/16 & 24]

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

4.4.1 [39/35+]

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 [43/20]

Add following paragraph as a NOTE

Examples of type definitions with type parameters are:

<<<<<<<<<<<<<<<<<<<< **Start of Text Option 1** >>>>>>>>>>>>>>>>>>>>

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

Here **`WP`** is implicitly defined to be a static parameter since it is used to determine the kind of **`comp`**. **`ORDER`** is implicitly nonstatic.

<<<<<<<<<<<<<<<<<<<< **End of Text Option 1** >>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<< **Start of Text Option 2** >>>>>>>>>>>>>>>>>>>>

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

Here **`WP`** is implicitly defined to be a static parameter since it is used to determine the kind of **`comp`**. **`ORDER`** is implicitly nonstatic.

<<<<<<<<<<<<<<<<<<<< **End of Text Option 2** >>>>>>>>>>>>>>>>>>>>

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 a vector in a 100 dimension space and each component is represented by a double precision real.

The definition of vector could equally well be written as:

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

where the nature of the parameters has been declared explicitly.

It would also be valid to declare **ORDER** to be **STATIC** in which case vectors of different orders could be used to resolve generic overloads.

4.1.2 Edits to implement parameter value inquiry

[45/23+]

Insert new section 4.5 and renumber subsequent sections:

<<<<<<<<<<<<<<<<<<<<Start of Text option 1>>>>>>>>>>>>>>>>>>>>

4.5 Type parameter value inquiry

Type parameter values may be obtained through a type parameter reference.

R431.1 *type-param-ref* **is** *object-name%param-name*

Constraint: A *type-param-ref* shall only be used if *object-name* is parameterized
Constraint: For objects of type **INTEGER** and **REAL** *param-name* shall be **KIND**
Constraint: For objects of type **CHARACTER**, *param-name* shall be **LEN**
Constraint: A *type-param-ref* shall not appear in a context which assigns it a value

<<<<<<<<<<<<<<<<<<<<End of Text option 1>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<Start of Text option 2>>>>>>>>>>>>>>>>>>>>

4.5 Type parameter value inquiry

For each type parameter specified in a derived type definition there is a generic inquiry function that has the same name as the type parameter. This function takes as its single nonoptional argument any object 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, for the objects, **rotation** and **steepest** of the **VECTOR** type defined previously **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.

<<<<<<<<<<<<<<<<<<<<**End of Text option 2**>>>>>>>>>>>>>>>>>>>>

4.1.3Edits to implement structure constructor

4.4.4 [44/37]

Replace "value of" by "value of the"

<<<<<<<<<<<<<<<<<<<<**Start of Text option 1**>>>>>>>>>>>>>>>>>>>>

4.4.4 [44/39]

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

<<<<<<<<<<<<<<<<<<<<**End of Text option 1**>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<**Start of Text option 2**>>>>>>>>>>>>>>>>>>>>

4.4.4 [44/39]

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

<<<<<<<<<<<<<<<<<<<<**End of Text option 2**>>>>>>>>>>>>>>>>>>>>

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 as there are parameters. 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 static type parameters, each corresponding *type-param-expr* must be an initialization expression.

4.4.4 [44/44]

After “component." 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 [44/45]

Replace “component values” with “component values and type parameters”

4.4.4 [45/6]

Add the following NOTE

An example of a constructor for a parameterized type is:

<<<<<<<<<<<<<<<<<<<<**Start of Text option 1**>>>>>>>>>>>>>>>>>>>>

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

<<<<<<<<<<<<<<<<<<<<**End of Text option 1**>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<**Start of Text option 2**>>>>>>>>>>>>>>>>>>>>

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

<<<<<<<<<<<<<<<<<<<<**End of Text option 2**>>>>>>>>>>>>>>>>>>>>

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

4.1.4Additional edits to implement constructors as generic functions

{ *The following edits make constructors generic function references. These edits are separated out from those in the previous section which just extend the current definition to include type parameters* **}**

4.4.4 [44/36]
After "corresponding" add "generic function reference that is a"

4.4.4 [44/39]
Replace "*expr-list*" with "*comp-expr-list*"

Add

R431.1	*comp-expr*	**is**	[*component-name*=]*expr*
R431.2	*type-param-expr*	**is**	[*type-param-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*.

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

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

<<<<<<<<<<<<<<<<<<<<**Start of Text option 1**>>>>>>>>>>>>>>>>>>>>

4.4.4 [44/40]

Replace “component” by “parameter and component”

4.4.4 [44/41]
Replace “components” by “parameters and components”

4.4.4 [44/41]
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).

<<<<<<<<<<<<<<<<<<<<**End of Text option 1**>>>>>>>>>>>>>>>>>>>>

<<<<<<<<<<<<<<<<<<<<**Start of Text option 2**>>>>>>>>>>>>>>>>>>>>

4.4.4 [44/40]

Replace “component” by “component parameter and”

4.4.4 [44/41]
Replace “components” by “parameters and components”

4.4.4 [44/41]

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).

<<<<<<<<<<<<<<<<<<<<**End of Text option 2**>>>>>>>>>>>>>>>>>>>>>

4.1.5Declaration of objects

5.1 [47/25]

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

5.1 [47/41]

Add constraint

Constraint: The *type-selector* must appear if the type is parameterized and must not appear otherwise.

[48/47]

Replace “(5.1.1.5)” by “(5.1.1.5), a nonstatic *type-param-value* (5.1.1.7), “

5.1.1.7 [52/8]

Add rules and constraints

R510.1 *type-selector* is (*type-param-selector-list*)

R510.2 *type-param-selector* is [*type-param-name*=]*type-param-expr*

R510.3 *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 static 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.

5.1.1.7 [52/16+]

Add paragraph

An assumed type parameter value is a nonstatic type parameter value of a derived type dummy argument that is specified with an asterisk *type-param-value*. Such a value may only be used to declare a dummy argument of a procedure, in which case the type parameter of the dummy argument assumes the value of the associated actual argument when the procedure is invoked.

5.5.2.3 [70/16]

After "type" add "and type parameters"

7.1.4.2 [92/17]

After the second "The type" add "and type parameters."

7.1.6.1 [93/25+ &94/17+]

Add new list item and renumber the next list item as (f)

(e) a derived-type static type parameter inquiry expression, or

7.1.6.2 [95/36]

Add new list item and renumber the next list item as (f)

(e) a derived-type type parameter inquiry expression"

7.1.7 [97/40]

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 static 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 [107/39]

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

7.5.1.2 [108/8]

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

11.3.2 [187/31+]

Add sentence

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

12.2.1.1 [192/17]

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

12.3.1.1 [193/18]

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

12.3.1.1 [193/22]

Add additional item to list and renumber list

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

12.4.1.1 [200/5]

Add sentence after “dummy argument.”

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

14.1.2 [275/38]

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