To: J3 J3/24-139r2

From: John Reid & Hidetoshi Iwashita & Malcolm Cohen

Subject: Syntax for generic subprograms

Date: 2024-June-27

References: N2217, 23-223r2, 23-244r1.

1. Introduction

===============

At its meeting Jun 12-16, 2023, WG5 decided to approve generic subprograms as described in N2217 for Fortran 202Y. Use cases for this are set out in N2217. Formal requirements were approved by J3 in 23-233r2 (revised in 24-147). Formal specifications were approved by J3 in 23-244r1 (revised in 24-48). Here we present syntax.

2. Syntax

=========

x01. The GENERIC <prefix-spec> in the <subroutine-stmt> or <function-stmt>

of a module subprogram or internal subprogram specifies the subprogram to be generic. Its name is a generic name and it defines one or more specific procedures with that generic name. Each dummy argument of a specific procedure has a single type, kind, and rank. The interface of each specific procedure is explicit.

If the name is already generic, the new specific procedures will be added to the existing set of specific procedures. Any two of these procedures must satisfy the rules of 15.4.3.4.5 to ensure that any reference is unambiguous.

Constraint: If a module subprogram is generic, it shall not have an internal subprogram that is generic.

Comment

This is a simplification to avoid nested generic subprograms, which would generate N\*\*2 specifics (most of which might not even be referenced); that would be an unnecessary burden on the processor.

Instead of nested generic subprograms, side-by-side generic subprograms (in the containing scoping unit) can be used, perhaps making some of them PRIVATE if they are module subprograms.

Constraint: A generic subprogram shall not have an alternate return.

Comment

This is to avoid extending an obsolescent feature.

There is no other technical reason.

Example 1

MODULE mod

CONTAINS

GENERIC SUBROUTINE my\_lift(x)

TYPE(INTEGER, REAL) :: x

...

END SUBROUTINE

END MODULE mod

Example 2

PROGRAM main

...

CONTAINS

GENERIC SUBROUTINE my\_lift(x)

TYPE(INTEGER, REAL) :: x

...

END SUBROUTINE

END PROGRAM main

Note: Only internal subprograms and module subprograms can have the GENERIC prefix, not external subprograms or interface bodies other than separate module procedure interface bodies.

x01a. The GENERIC <prefix-spec> in the <subroutine-stmt> or <function-stmt> of a separate module procedure interface declares that the separate module procedure name is generic, and defined by a module subprogram that has both the GENERIC and MODULE prefixes.

Comment

The mp-subprogram-stmt ("MODULE PROCEDURE name") is not available in this case, as

(a) that would have no indication that the separate module subprogram is generic, and

(b) if there were more than one generic subprogram interface with the same name, it would be ambiguous which one it was supposed to be implementing.

x02. A <specific-procedure-list> in the PROCEDURE statement of a generic interface block or in the GENERIC statement is extended to specify generic names, as follows.

<specific-procedure> <<is>> <procedure-name>

<<or>> <generic-name>

A generic name appearing in the <specific-procedure-list> is treated as if all specific procedures identified by the generic name were added to the list.

Constraint: If a <generic-name> appears in the <specific-procedure-list> of a PROCEDURE statement in a generic interface block, the <generic-spec> of the <interface-stmt> shall not be a generic name. If a <generic-name> appears in the <specific-procedure-list> of a GENERIC statement, the <generic-spec> shall not be a generic name.

Comment

This constraint prohibits a generic name identifying generic names. It may cause complicated situations due to mutual inclusions, recurrent references, and evaluation order issues of generic names.

Invalid Example

MODULE bad

INTERFACE invalid

PROCEDURE xyz

PROCEDURE gsub ! Violates the above constraint.

END INTERFACE

CONTAINS

SUBROUTINE xyz()

END SUBROUTINE

GENERIC SUBROUTINE gsub(a)

TYPE(integer,real) :: a

a = 999

END SUBROUTINE

END MODULE

Example

MODULE example

INTERFACE OPERATOR(.myop.)

PROCEDURE fun ! All specific procedures with generic name fun

FUNCTION fen(a,b) ! External function fen

REAL, INTENT(IN) :: a, b

REAL :: fen

END FUNCTION fen

END INTERFACE

CONTAINS

GENERIC FUNCTION fun(a)

REAL, INTENT(IN), RANK(0) :: a

REAL, RANK(0) :: fun

...

END FUNCTION fun

GENERIC FUNCTION fun(a) RESULT(b)

REAL, INTENT(IN), RANK(1:) :: a

REAL, RANK(RANK(a)) :: b

...

END FUNCTION fun

END MODULE

x03. A generic dummy argument is a type-or-kind-generic dummy argument, and/or a rank-generic dummy argument. A type-or-kind-generic dummy argument is declared with a <generic-type-spec>. A rank-generic dummy argument is declared with a <generic-rank-spec>.

NOTE:

An entity (which may be a dummy argument) whose type, kind, or rank depends on those of a generic dummy argument is not itself a generic dummy argument, but is a <<generic-dependent entity>>. The syntax for these will be discussed later.

A <generic-type-spec> or <generic-rank-spec> can appear only in a "generic type declaration statement" <generic-type-decl-stmt>. A <generic-type-decl-stmt> is a <specification-construct> that shall appear only in the specification part of a generic subprogram.

R8nn generic-type-decl-stmt is generic-type-spec

[ [ , generic-attr-spec ] ... :: ]

generic-dummy-arg-decl-list

or declaration-type-spec

, generic-attr-spec-list ::

generic-dummy-arg-decl-list

C8nn If a generic-type-decl-stmt does not have a generic-type-spec, its generic-attr-spec-list shall contain a generic-rank-spec.

R8nn generic-dummy-arg-decl is dummy-arg-name [ ( array-spec ) ]

[ \* char-length ]

C8nn A generic dummy argument shall not be a coarray.

Comment

Codimension is not mentioned in the requirements, use cases, or specifications, but seems like it would add complication. So we should prohibit it for now.

C8nn A generic dummy argument shall be a nonoptional dummy data object.

Comment

Non-optionality is absolutely required, otherwise the generated specific procedures would be ambiguous.

\*\*\*ADDITIONAL FORMAL SPECIFICATIONS

sNN A generic dummy argument cannot be a coarray.

Reason: This is just a simplification. At a future time we could allow this, and we could also allow generic-corank dummy arguments.

sNN A generic dummy argument shall be a dummy data object not a procedure.

Reason: This is another simplification. Syntactically, it would need to allow a generic (type/kind/rank) specification to appear in an interface body in a generic subprogram.

\*\*\*END ADDITIONAL FORMAL SPECIFICATION

Now we come to the question of dummy procedures.

There are three possibilities here, which I will list from the most stringent (greatest simplifying) to the least.

(1) Constraint: A generic subprogram shall not have a dummy procedure.

(2) Constraint: A dummy procedure of a generic subprogram shall have an explicit interface, and shall not be a generic dummy argument or a generic-dependent entity.

(3) Constraint: A dummy procedure of a generic subprogram shall have an explicit interface, and shall not be a generic dummy argument, but may be a generic-dependent entity.

Comment

(a) Requiring an explicit interface is a simplification that also improves safety.

(b) We already constrained against a procedure being a generic dummy argument above.

(c) Simplification (1) seems to be overly strict (there can surely be no problem with callbacks that do not involve genericity). Simplification (2) is therefore the minimum we should consider, but that would rule out functions whose argument or result type depend on a generic dummy argument, and that would be a loss of functionality.

PROPOSED: Let's go with option (3) for now.

x04. Syntax for kind-generic specifications.

R8nn kind-generic-type-spec is generic-intrinsic-type-spec

or generic-derived-type-spec

R8nn generic-intrinsic-type-spec is nonchar-intrinsic-type-name

( [ KIND = ] int-constant-expr )

or CHARACTER ( gen-char-type-params )

R8nn nonchar-intrinsic-type-name is REAL | INTEGER | LOGICAL | COMPLEX

R8nn gen-char-type-params is gen-char-len [ KIND = ] int-constant-expr

or LEN= gen-char-len KIND= int-constant-expr

or KIND= int-constant-expr LEN= gen-char-len

R8nn gen-char-len is \* | :

C8nn The int-constant-expr in a generic-intrinsic-type-spec shall be

an array of rank one.

Comment

Basically, a kind-generic type-spec looks like a normal one except that it has an array expression for the kind type parameter.

Example x04-1

GENERIC SUBROUTINE gensub(x,y)

INTEGER([int8,int16,int32]),INTENT(INOUT) :: x

CHARACTER(\*,KIND=[ascii,iso\_10646]) :: y

R8nn generic-derived-type-spec is type-name ( gen-tp-spec-list )

R8nn gen-tp-spec is [ keyword = ] gen-tp-value

R8nn gen-tp-value is int-constant-expr | \* | :

C8nn A gen-tp-value shall be \* or : if and only if the type parameter is a length type parameter, otherwise the int-constant-expr shall be an array of rank one.

C8nn A generic-derived-type-spec shall specify at least one kind type parameter.

Example x04-2

TYPE T(k1,k2,n)

INTEGER,KIND :: k1,k2

INTEGER,LEN :: n

REAL(k1) value(k2,n)

END TYPE

GENERIC SUBROUTINE gensub2(x)

TYPE(t([kind(0.0),kind(0d0)],k2=[1,2,4,8],n=\*)),INTENT(INOUT) :: x

This covers 2 values for k1, and independently 4 values for k2, thus eight specifics:

TYPE(t(k1=kind(0.0),k2=1,n=\*)

TYPE(t(k1=kind(0.0),k2=2,n=\*)

TYPE(t(k1=kind(0.0),k2=4,n=\*)

TYPE(t(k1=kind(0.0),k2=8,n=\*)

TYPE(t(k1=kind(0d0),k2=1,n=\*)

TYPE(t(k1=kind(0d0),k2=2,n=\*)

TYPE(t(k1=kind(0d0),k2=4,n=\*)

TYPE(t(k1=kind(0d0),k2=8,n=\*)

\*\*\*BEGIN OPTIONAL

x04a. Have special syntax, for intrinsic types only, that mean all the possible kind values.

R8nn generic-intrinsic-type-spec is nonchar-intrinsic-type-name

( [ KIND = ] generic-kind )

or nonchar-intrinsic-type-name

( [ KIND = ] \* )

or CHARACTER ( gen-char-type-params )

R8nn nonchar-intrinsic-type-name is REAL | INTEGER | LOGICAL | COMPLEX

R8nn gen-char-type-params is gen-char-len [ KIND = ] generic-kind

or LEN= gen-char-len KIND= generic-kind

or KIND= generic-kind LEN= gen-char-len

R8nn generic-kind is int-constant-expr

or \*

Comment

This is optional because REAL(\*) would have identical effect to REAL(real\_kinds), similarly INTEGER(\*) and INTEGER(integer\_kinds), etc. Perhaps the \* looks better though? Plus one does not need to use ISO\_FORTRAN\_ENV to get them.

\*\*\*END OPTIONAL

\*\*\*BEGIN OPTIONAL 2

x04b. Should we permit duplicate values (and ignore duplicates)?

Yes: normative text

"Duplicate kind values in a kind-generic-type-spec are permitted, and treated as if only one appeared."

No:

C8nn Kind values specified in a kind-generic-type-spec shall be distinct.

For example, maybe the user wants to specify

TYPE(REAL([selected\_real\_kind(3),selected\_real\_kind(2)])) x

On a processor with no 16-bit real, or which has only one kind of 16-bit real, this would have duplicate values; on a processor that has both IEEE 16-bit and bfloat16, the values would be distinct.

\*\*\*END OPTIONAL 2

x05. Syntax for type-generic specifications.

R8nn generic-type-spec is TYPE ( generic-type-specifier-list )

or CLASS ( generic-type-specifier-list )

or generic-intrinsic-type-spec

R8nn generic-type-specifier is intrinsic-type-spec

or derived-type-spec

or enum-type-spec

or enumeration-type-spec

or kind-generic-type-spec

C8nn If the generic-type-spec keyword is CLASS, each generic-type-specifier shall identify an extensible type.

C8nn A generic-type-specifier-list that contains no kind-generic-type-spec shall have more than one item.

C8nn A generic-type-specifier shall specify that each length type parameter is assumed or deferred.

Comment

Length type parameters do not participate in generic resolution, so this simplification is mostly about saving the user's toes.

Example x05-1

GENERIC FUNCTION plus(a,b) RESULT(r)

TYPE(integer,real,complex),INTENT(IN) :: a

TYPEOF(a),INTENT(IN) :: b

TYPEOF(a) :: r

r = a + b

END FUNCTION

This will generate three specifics, with signatures

integer function(integer a,b)

real function(real a,b)

complex function(complex a,b)

Example x05-2

TYPE t1

...

END TYPE

TYPE t2

...

END TYPE

GENERIC SUBROUTINE process(x)

CLASS(t1,t2),INTENT(IN) :: x

...

END SUBROUTINE

This will generate two specifics, one with a CLASS(t1) argument, the other with a CLASS(t2) argument.

\*\*\*BEGIN OPTIONAL 3

x05a. Should we permit duplicate type+kind here?

Here is the example that argues for "Yes".

USE ISO\_FORTRAN\_ENV

TYPE(REAL(REAL64), DOUBLE PRECISION) :: x

TYPE(INTEGER, INTEGER(INT32)) :: y

On some processors, the first type-spec will specify the same type and type parameters as the second in one of those statements, and on other processors the types will be distinct.

Yes: normative text

"A generic-type-specifier in a generic type declaration statement may specify the same type and type parameters as another. The redundant specification is ignored; however, the dummy arguments remain generic dummy arguments."

No: constraint

C8nn A generic-type-specifier in a generic-type-decl-stmt shall not specify the same type and type parameters as another generic- type-specifier in that statement.

\*\*\*END OPTIONAL 3

x06. Rank-generic specification

A dummy argument that is generic by rank is declared using a generic- rank-spec; having a generic-rank-spec makes a type declaration stmt into a generic type declaration statement, even if it is not generic by type/kind.

R8nn generic-rank-spec is RANK ( generic-rank-list )

R8nn generic-rank is scalar-int-constant-expr

or generic-rank-range

or \*

R8nn generic-rank-range

is scalar-int-constant-expr : scalar-int-constant-expr

C8nn A scalar-int-constant-expr in a generic-rank-list shall be

nonnegative.

C8nn If a generic-rank-list contains no \* or generic-rank-range, it

shall contain at least two <scalar-int-constant-expr>s.

\*\*\*BEGIN OPTIONAL

May want to avoid \* for generic-rank as RANK(\*) normally means assumed-size.

\*\*\*END OPTIONAL

A generic-rank-range specifies all the values that are both greater than or equal to the first expression and less than or equal to the second expression.

A generic-rank that is \* specifies all ranks that are supported by the processor.

C8nn A generic-rank-list shall not specify the same rank more than once.

Comment

It does not seem useful to permit duplicates here.

\*\*\*BEGIN OPTIONAL 4

If we wish to permit duplicate ranks, normative text

"Duplicate values specified by a generic-rank-list are permitted; the duplicate values are ignored. If that means that only one rank is applicable, the dummy argument still remains generic."

\*\*\*END OPTIONAL 4

Example x06

GENERIC SUBROUTINE lift(x,y)

TYPE(INTEGER(int32,int64), REAL), RANK(1:2), ALLOCATABLE :: x, y

TYPEOF(x),RANKOF(y),ALLOCATABLE :: z

...

END SUBROUTINE

This subroutine defines 36 specific procedures with generic name lift, in which the variables x, y and z are allocatable, and have other characteristics respectively:

int32, rank 1 int32, rank 1 int32, rank 1

int64, rank 1 int32, rank 1 int64, rank 1

real, rank 1 int32, rank 1 real, rank 1

int32, rank 2 int32, rank 1 int32, rank 1

int64, rank 2 int32, rank 1 int64, rank 1

real, rank 2 int32, rank 1 real, rank 1

int32, rank 1 int32, rank 2 int32, rank 2

... etc.

Example x06-2

GENERIC SUBROUTINE lift(x,y)

TYPE(INTEGER(int32,int64), REAL), RANK(1:2), ALLOCATABLE :: x

TYPEOF(x), RANKOF(x), ALLOCATABLE :: y

TYPEOF(x),RANKOF(y),ALLOCATABLE :: z

...

END SUBROUTINE

This subroutine defines only 6 specific procedures with generic name

lift.

x07. When specifying generic rank, it may be desirable to be able to specify all valid ranks greater than a minimum value, e.g. all arrays of rank two or more. There are two possibilities:

(1) permit rank specification higher than the maximum supported rank, and ignore them;

(2) provide syntax or a named constant to represent the maximum supported rank - syntax could be to allow "\*" in a generic-rank-range, a named constant could be MAX\_RANK.

PROPOSAL: Add a new named constant to ISO\_FORTRAN\_ENV, MAX\_RANK, whose value is the maximum supported rank for a noncoarray.

x08. A local entity, including a dummy argument and function result, may be declared to have the type, kind, or rank of a previously declared generic dummy argument. Such an entity is a generic-dependent entity, and will have the same type, kind, and/or rank as the generic dummy argument(s) on which it depends. This can be achieved with the TYPEOF and CLASSOF type specifiers, the KIND function applied to a generic dummy argument, and the (new) RANKOF clause.

(additional) R802 attr-spec ...

or RANKOF (dummy-argument-name)

C8nn The dummy-argument-name in a RANKOF clause shall be the name of a generic dummy argument.

C8nn If the RANKOF clause appears in an attr-spec-list, the RANK or DIMENSION clause shall not appear.

C8nn The rank of an entity shall not depend on the rank of a generic dummy argument other than via the RANKOF clause.

Example

GENERIC SUBROUTINE lift(x,y)

TYPE (INTEGER, REAL), ALLOCATABLE, RANK(1:2) :: x

TYPEOF(x), RANKOF(x) :: y

...

END SUBROUTINE

x09. The SELECT GENERIC RANK construct in a generic subprogram selects at most one of its constituent blocks in each instance of the subprogram.

R1150a <select-grank-construct> <<is>> <select-grank-stmt>

[ <select-grank-case-stmt>

<block> ]...

<end-select-grank-stmt>

R1151a <select-grank-stmt> <<is>>

[ <select-construct-name> : ] SELECT GENERIC RANK ( <selector> )

C1155a The <selector> in a <select-grank-stmt> shall be a named

generic dummy argument that is generic by rank.

R1152a <select-grank-case-stmt>

is RANK ( <generic-rank-list> ) [ <select-construct-name> ]

or RANK DEFAULT [ <select-construct-name> ]

C1152a A generic-rank in a select-grank-case-stmt shall not be \*.

<generic-rank-list> is defined in x07.

Duplicate values are treated the same here as there (see OPTIONAL 4).

C1157a For a given <select-grank-construct>, the same value shall not be specified in more than one <select-grank-case-stmt>.

C1158a For a given <select-grank-construct>, there shall be at most one RANK DEFAULT <select-grank-case-stmt>.

R1153a <end-select-grank-stmt> <<is>>

END SELECT [ <select-construct-name> ]

The rules on <select-construct-name> are similar to those of the SELECT CASE construct.

The execution of a SELECT GENERIC RANK construct with SELECT GENERIC RANK statement:

SELECT GENERIC RANK ( x )

is similar to that of a SELECT CASE construct with SELECT CASE statement:

SELECT CASE ( RANK(x) )

where the rank of x is constant.

Comment. The intention is that each instantiation shall contain executable code for at most one block. Unlike the SELECT CASE construct, the SELECT GENERIC RANK statement is expected to have no runtime overhead of selection and branching.

Example

GENERIC FUNCTION fun(x) RESULT(y)

TYPE(type1), RANK(0:7) :: x

TYPEOF(x), RANK(RANK(x)) :: y

SELECT GENERIC RANK (y)

RANK (0)

!! code if y is a scalar

RANK (1:3)

!! code if y is an array of 1 to 3 dimensions

RANK DEFAULT

!! code if y is an array of 4 to 7 dimensions

END SELECT

END FUNCTION fun

x10. The SELECT GENERIC TYPE construct in a generic subprogram selects at most one of its constituent blocks in each instance of the subprogram.

Comment. The selection is based on the declared type of a generic-type or generic-kind entity, not the dynamic type.

R1154a <select-gtype-construct> <<is>> <select-gtype-stmt>

[ <gtype-guard-stmt>

<block> ]...

<end-select-gtype-stmt>

R1155a <select-gtype-stmt> <<is>>

[ <select-construct-name> : ] SELECT GENERIC TYPE ( <selector> )

C1163a The <selector> in a <select-gtype-stmt> shall be a generic

dummy argument that is generic by type or kind.

R1156a <gtype-guard-stmt>

<<is>> TYPE IS ( <type-spec> ) [ <select-construct-name> ]

<<or>> TYPE DEFAULT [ <select-construct-name> ]

C1165a The <type-spec> shall be <intrinsic-type-spec> or

<derived-type-spec> and shall specify that each length type

parameter is assumed.

C1168a For a given <select-gtype-construct>, the same type and kind

type parameter values shall not be specified in more than

one TYPE IS <type-guard-stmt>.

C1169a For a given <select-type-construct>, there shall be at most

one CLASS DEFAULT <type-guard-stmt>.

R1157a <end-select-gtype-stmt> <<is>>

END SELECT [ <select-construct-name> ]

Execution:

A SELECT GENERIC TYPE construct selects just one block to be executed. Regardless of whether the selector is polymorphic or not, the block to be executed is selected by the declared type and kind of the <selector>. If it matches the <type-spec> of a TYPE IS <gtype-guard-stmt>, the block following that statement is selected. Otherwise, if there is a TYPE DEFAULT <gtype-gurard-stmt>, the block following that statement is selected, else, no block is selected.

Comment: The block is selected at compile time. The execution of the SELECT GENERIC TYPE construct is expected to be the execution

of the selected block if any.

Example

GENERIC FUNCTION fun(x) RESULT(y)

TYPE(type1,type2) :: x, y

!! code if x is type1 or type2

SELECT GENERIC TYPE (x)

TYPE IS (type1)

!! code if x is type1

TYPE IS (type2)

!! code if x is type2

END SELECT

END FUNCTION fun

3. Further Examples

===================

Example 1

MODULE example

INTERFACE OPERATOR(.myop.)

PROCEDURE s ! All of the specific procedures of s.

END INTERFACE

CONTAINS

GENERIC FUNCTION s(a,b) RESULT(c)

TYPE(REAL,COMPLEX), INTENT(IN), RANK(\*) :: a

TYPEOF(a),RANKOF(a), INTENT(IN) :: b

TYPEOF(a), RANKOF(a) :: c, temp

...

SELECT GENERIC TYPE (a)

TYPE IS (REAL)

temp = temp \* (1-b)

TYPE IS (COMPLEX)

! Just this once, we want the conjugate.

temp = temp \* (1-CONJG(b))

END SELECT

...

c = temp

END FUNCTION

END MODULE

Example 2

PROGRAM main

INTEGER:: n = 5

WRITE(\*,\*) factorial(n)

CONTAINS

GENERIC RECURSIVE FUNCTION factorial(n) RESULT (res)

USE ISO\_FORTRAN\_ENV

INTEGER (int\_kinds) :: n

TYPEOF(n) :: res

IF (n >1) THEN

res = n\*factorial(n-1)

ELSE IF (n==1) THEN

res = 1

ELSE

res =0

END IF

END FUNCTION factorial

END PROGRAM main