Proposal for Generic Subprogram

Version 1.4
Hideoshi Iwashita
July 31, 2023

1. Introduction

The mechanism of a generic identifier for selecting specific procedures is an outstanding feature of Fortran. A generic identifier (generic name, operator, or assignment) identifies one of the specific procedures whose argument types, kinds, or ranks differ from each other. In Fortran, most intrinsic procedures and operators are generic. For example, the arguments of the intrinsic function MAX can be integer, real, or character types, and the operands of the operator + can be integer, real, or complex types. It is a natural and productive programming style to use generic names and operators. The same is true for user-defined procedures and derived types.

Importantly, using a generic identifier should not affect execution performance. Not compromising performance is an essential requirement in Fortran. The generic identifier mechanism achieves it through the following dedicate considerations:

- Selecting a specific procedure depends only on static parameters and is determined at compile time. Therefore, no overhead of judgment or branching remains on the runtime code.
- Since the generic identifier is resolved within or before the compiler front-end, it does not affect the existing sophisticated optimization and code generation within the compiler back-end.

Thus, it can be said that generic identifier mechanism is a feature that combines convenience and performance for users. Whereas library providers who create specific procedures and publish them as a generic identifier still face a major challenge: combinational explosion. As programmers attempt to generalize the types and ranks of library procedures, the number of specific subprograms can grow enormously, into the tens or hundreds. For example, to define a function whose argument variable has any arithmetic type (integer, real or complex with any kind parameter) and any rank (0 through 15 in standard), the programmer must write totally more than 100 specific function subprograms. Even if such a huge number of specific subprograms could be written using clever editors and tools, maintaining and improving such a number of versions is error-prone and a waste of time.

This paper proposes an extension of the generic identifier mechanism to easily define large numbers of specific procedures. Instead of writing a large number of subprograms, the user only needs to write a **generic subprogram** that defines multiple specific procedures.

In this paper, Section 2 demonstrates examples for quick understanding at first, Section 3 describes the syntax, and Section 4 summarizes.

2. Example

Consider a simple function that returns true if the argument is a NaN (not a number) or has at least one NaN array element, and false otherwise. The argument is allowed to be a variable of 32, 64, or 128-byte real type with rank from 0 to 15.

2.1 Original set of specific functions

List 1 shows an example of defining generic function `has_nan` with 48 specific functions for all types and all ranks. As you can see, most of the functions have the same body, but since they have different ranks or different kind parameters from each other, they must be written as separate functions in the current Fortran standard.

List 1. has_nan defined by specific subprograms

```
MODULE mod_nan_original
   USE :: ieee_arithmetic
   USE :: iso_fortran_env
   IMPLICIT NONE

   INTERFACE has_nan
      MODULE PROCEDURE :: &
         has_nan_r32_0,   has_nan_r32_1,   has_nan_r32_2,   has_nan_r32_3, &
         has_nan_r32_4,   has_nan_r32_5,   has_nan_r32_6,   has_nan_r32_7, &
         has_nan_r32_8,   has_nan_r32_9,   has_nan_r32_10,  has_nan_r32_11, &
         has_nan_r32_12,  has_nan_r32_13,  has_nan_r32_14,  has_nan_r32_15, &
         has_nan_r64_0,   has_nan_r64_1,   has_nan_r64_2,   has_nan_r64_3, &
         has_nan_r64_4,   has_nan_r64_5,   has_nan_r64_6,   has_nan_r64_7, &
         has_nan_r64_8,   has_nan_r64_9,   has_nan_r64_10,  has_nan_r64_11, &
         has_nan_r64_12,  has_nan_r64_13,  has_nan_r64_14,  has_nan_r64_15, &
         has_nan_r128_0,  has_nan_r128_1,  has_nan_r128_2,  has_nan_r128_3, &
         has_nan_r128_4,  has_nan_r128_5,  has_nan_r128_6,  has_nan_r128_7, &
         has_nan_r128_8,  has_nan_r128_9,  has_nan_r128_10, has_nan_r128_11, &
         has_nan_r128_12, has_nan_r128_13, has_nan_r128_14, has_nan_r128_15
   END INTERFACE has_nan

   PRIVATE
   PUBLIC :: has_nan

 CONTAINS

   FUNCTION has_nan_r32_0(x) RESULT(ans)
     REAL(REAL32), INTENT(IN) :: x
     LOGICAL :: ans
     ans = ieee_is_nan(x)
   END FUNCTION has_nan_r32_0

   FUNCTION has_nan_r32_1(x) RESULT(ans)
     REAL(REAL32), INTENT(IN) :: x(:)
     LOGICAL :: ans
     ans = any(ieee_is_nan(x))
   END FUNCTION has_nan_r32_1

   ... (omit 65 lines of code)

   FUNCTION has_nan_r32_15(x) RESULT(ans)
     REAL(REAL32), INTENT(IN) :: x(:,:,:,:,:,:,:,:,:,:,:,:,:,:,:)
     LOGICAL :: ans
     ans = any(ieee_is_nan(x))
   END FUNCTION has_nan_r32_15

   ... (omit 155 lines of code)

   FUNCTION has_nan_r128_15(x) RESULT(ans)
     REAL(REAL128), INTENT(IN) :: x(:,:,:,:,:,:,:,:,:,:,:,:,:,:,:)
     LOGICAL :: ans
     ans = any(ieee_is_nan(x))
   END FUNCTION has_nan_r128_15

END MODULE mod_nan_original
```

2.2 Generic subprogram

List 2 shows the equivalent code to the code of List 1, written using the generic subprogram proposed in this paper. A subprogram with the GENERIC prefix is a generic subprogram. The first generic subprogram defines three specific procedures where `x` is one of real types of 32, 64, and 128 bytes, respectively. The second generic subprogram defines 3 × 15 specific procedures where `x` is one of the combinations of 32, 64, or 128-byte real types and ranks from 1 to 15, respectively. Every specific procedure defined by the generic subprogram has no name and is referenced by the generic name.

List 2. has_nan defined with generic subprogram

```
MODULE mod_nan_proposed
  USE :: ieee_arithmetic
  USE :: iso_fortran_env

  PRIVATE
  PUBLIC :: has_nan

CONTAINS

  GENERIC FUNCTION has_nan(x) RESULT(ans)
    REAL(REAL32,REAL64,REAL128), RANK(0), INTENT(IN) :: x
    LOGICAL :: ans
    ans = ieee_is_nan(x)
  END FUNCTION has_nan

  GENERIC FUNCTION has_nan(x) RESULT(ans)
    REAL(REAL32,REAL64,REAL128), RANK(1:15), INTENT(IN) :: x
    LOGICAL :: ans
    ans = any(ieee_is_nan(x))
  END FUNCTION has_nan

END MODULE mod_nan_proposed
```

Multiple specific subprograms that have the same body except for type declaration statements for the dummy arguments can be combined into one generic subprogram. This may greatly reduce the amount of program code. In addition, since the generic subprogram is expanded to a list of the corresponding specific procedures, there should be no performance degradation.

3. Syntax

A **generic subprogram** is a subprogram that has the GENERIC prefix (3.1), which defines one or more specific procedures that have dummy arguments of different types, kinds, or ranks from each other. The name of a generic subprogram is a generic name for all defined specific procedures. Each specific procedure does not have a specific name.

A **generic type declaration statement** is the type declaration statement that specifies at least one dummy argument of the generic subprogram, which is extended to specify alternative types, kinds, and ranks (3.2).

3.1 GENERIC prefix

The GENERIC prefix of a FUNCTION or SUBROUTINE statement specifies that the subprogram is a generic subprogram.

The *prefix-spec*, the *function-stmt*, and the *subroutine-stmt* (F2023:15.6.2.1-3) are extended as follows.

R1530x *prefix-spec*	**is**	*declaration-type-spec*	**or**	ELEMENTAL	**or**	IMPURE
	or	MODULE	**or**	NON_RECURSIVE	**or**	PURE
	or	RECURCIVE	**or**	SYMPLE	**or**	**GENERIC**

R1533x *function-stmt* **is** [*prefix*] FUNCTION ***function-spec***
([*dummy-arg-name-list*]) [*suffix*]

R1533a ***function-spec*** **is** *function-name*
or ***generic-spec***

Constraint: The *function-spec* shall be *generic-spec* if the GENERIC prefix appears in the *prefix* and shall be *function-name* otherwise.

R1538x *subroutine-stmt* **is** [*prefix*] SUBROUTINE ***subroutine-spec***
[([*dummy-arg-list*]) [*proc-language-binding-spec*]]

Constraint: If the GENERIC prefix appears in the *prefix*, the *proc-language-binding-spec* shall not appear.

R1538a ***subroutine-spec*** **is** *subroutine-name*
or ***generic-spec***

Constraint: The *subroutine-spec* shall be *generic-spec* if the GENERIC prefix appears in the *prefix* and shall be *subroutine-name* otherwise.

R1508(asis) ***generic-spec***	**is**	*generic-name*
	or	OPERATOR (*defined-operator*)
	or	ASSIGNMENT (=)
	or	*defined-io-generic-spec*
R1509(asis) *defined-io-generic-spec*	**is**	READ (FORMATTED)
	or	READ (UNFORMATTED)
	or	WRITE (FORMATTED)
	or	WRITE (UNFORMATTED)

Add the following constraint to *interface-block* (F2023: R1501).

Constraint: If a generic subprogram appears as a constituents of an *interface-block*, the *interface-block* shall be generic and its *generic-spec* shall be identical to the *generic-spec* of the generic subprogram.

NOTE 1

The following is an example of a module that has generic function subprograms as the module subprograms.

```
MODULE M_ABSMAX

CONTAINS

  GENERIC FUNCTION ABSMAX(X) RESULT(Y)
    TYPE(INTEGER,REAL,DOUBLE PRECISION) :: X(:)
    TYPEOF(X) :: Y

    Y = MAXVAL(ABS(X))
    RETURN
  END FUNCTION ABSMAX

  GENERIC FUNCTION ABSMAX(X) RESULT(Y)
    COMPLEX :: X(:)
    REAL :: Y

    Y = MAXVAL(ABS(X))
    RETURN
  END FUNCTION ABSMAX

END MODULE M_ABSMAX
```

Where `TYPE(INTEGER,REAL,DOUBLE PRECISION)` specifies that `X` is an integer, real, or double precision type for each specific procedure (3.2.1). Two module subprograms are generic and specify the same generic name. Since their interfaces are explicit, they can be referenced in the host and sibling scopes. Therefore, the above program is equivalent to the following program.

```
MODULE M_ABSMAX

  INTERFACE ABSMAX
    MODULE PROCEDURE :: ABSMAX_I, ABSMAX_R, ABSMAX_D, ABSMAX_Z
  END INTERFACE

  PRIVATE
  PUBLIC :: ABSMAX

CONTAINS

  FUNCTION ABSMAX_I(X) RESULT(Y)
    TYPE(INTEGER) :: X(:)
    TYPEOF(X) :: Y

    Y = MAXVAL(ABS(X))
    RETURN
  END FUNCTION ABSMAX_I

  FUNCTION ABSMAX_R(X) RESULT(Y)
    TYPE(REAL) :: X(:)
    TYPEOF(X) :: Y

    Y = MAXVAL(ABS(X))
    RETURN
  END FUNCTION ABSMAX_R

  FUNCTION ABSMAX_D(X) RESULT(Y)
    TYPE(DOUBLE PRECISION) :: X(:)
    TYPEOF(X) :: Y

    Y = MAXVAL(ABS(X))
    RETURN
  END FUNCTION ABSMAX_D

  FUNCTION ABSMAX_Z(X) RESULT(Y)
    COMPLEX :: X(:)
    REAL :: Y

    Y = MAXVAL(ABS(X))
    RETURN
  END FUNCTION ABSMAX_Z

END MODULE M_ABSMAX
```

NOTE 2

Generic subprograms can be external. The following shows an interface block for ABSMAX if two module generic functions in NOTE 1 would be external.

```
INTERFACE ABSMAX
   GENERIC FUNCTION ABSMAX(X) RESULT(Y)
     TYPE(INTEGER,REAL,DOUBLE PRECISION) :: X(:)
     TYPEOF(X) :: Y
   END FUNCTION ABSMAX

   GENERIC FUNCTION ABSMAX(X) RESULT(Y)
     COMPLEX :: X(:)
     REAL :: Y
   END FUNCTION ABSMAX
END INTERFACE ABSMAX
```

NOTE 3

The following example shows a generic subprogram that defines an operator.

```
MODULE coord_m
  USE iso_fortran_env

  TYPE coord_t(k)
     INTEGER, KIND :: k
     REAL(kind=k) :: x, y, z
  END TYPE coord_t

CONTAINS

  GENERIC FUNCTION OPERATOR(+)(a, b) RESULT(c)
    TYPE(coord_t(real32,real64)), INTENT(IN) :: a, b
    TYPEOF(a) :: c

    c%x = a%x + b%x
    c%y = a%y + b%y
    c%z = a%z + b%z
    RETURN
  END FUNCTION OPERATOR(+)

END MODULE coord_m
```

The type `coord_t` has components `x`, `y`, and `z` of real type whose common kind is parameterized. The generic subprogram defines + operations between `coord_t(real32)` objects and between `coord_t(real64)` objects.

NOTE 4

The following example shows a generic subprogram that defines a defined I/O procedure.

```
GENERIC SUBROUTINE WRITE(FORMATTED) (data, unit, iotype, v_list, iostat, iomsg)
  class(coord_t(real32,real64)), intent(in) :: data
  integer, intent(in) :: unit
  character(*), intent(in) :: iotype
  integer, intent(in) :: v_list(:)
  integer, intent(out) :: iostat
  character(*), intent(inout) :: iomsg

  character(10) :: dedit
  character(100) :: formt

  write(dedit, '( "F", I0, ".", I0 )') v_list(1), v_list(2)
  formt = "('[ '," // dedit // ",', '," // dedit // ",', '," // dedit // ",' ]')"
  write(unit, fmt=formt, iostat=iostat) data%x, data%y, data%z
END SUBROUTINE WRITE(FORMATTED)
```

The generic subprogram defines a behavior of the DT edit descriptor in the formatted WRITE statement for types `coord_t(real32)` and `coord_t(real64)`. Using this generic subprogram, the following code works:

```
type(coord_t(real32)) :: cod32
type(coord_t(real64)) :: cod64

cod32%x = 1.1111111111111111111d0
cod32%y = 2.2222222222222222222d0
cod32%z = 3.3333333333333333333d0
write(*, "(DT(20,17))") cod32

cod64%x = 1.1111111111111111111d0
cod64%y = 2.2222222222222222222d0
cod64%z = 3.3333333333333333333d0
write(*, "(DT(20,17))") cod64
```

The example of the result is shown below:

```
[  1.11111116409301758,  2.22222232818603516,  3.33333325386047363 ]
[  1.11111111111111116,  2.22222222222222232,  3.33333333333333348 ]
```

Comment:

- Constraints for the interface block seems not sufficient. It should be summarized in another section.
- Specific procedure names are undefined. Do we need to identify the specific procedures by name or in some other way? If so, how can it be specified?
  - An actual argument can be a procedure name, which must be a specific name. Should we have a notation such as “ABSMAX when the first argument is the default real type”?
  - There seems to be a need to call generic procedures from C language. Is there a need to extend the BIND statement for this case? For example,

```
BIND (C, NAME="c_name", ARGS=("float","char[10]")) :: generic_name
```

3.2 Extension of the type declaration statement

The type declaration statement is defined as follows in Fortran 2023:

R801(asis) *type-declaration-statement* **is** *declaration-type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*

The *declaration-type-spec* and the *attr-spec* are extended to specify alternative types (3.2.1), kinds (3.2.2, 3.2.3), and ranks (3.2.4).

Constraint: If a *type-declaration-statement* has alternative types or kinds, at least one entity in the *entity-decl-list* shall be a dummy argument.

Constraint: If a *type-declaration-statement* has alternative ranks, at least one entity in the *entity-decl-list* shall be a dummy argument that does not have an *array-spec*.

NOTE 1

The *declaration-type-spec* appearing in a *data-component-def-stmt* (F2023:R737), the prefix of a *function-stmt* (F2023:R1529), or an *implicit-spec* (F2023:R867) do not specify alternative types or kinds for entities in the *entity-decl-list*.

3.2.1 Alternative type specifier

The *declaration-type-spec* has been extended to have alternative types.

R703x	*declaration-type-spec*	**is**	*intrinsic-type-spec*
		or	TYPE (***alter-type-spec***)
		or	CLASS (***alter-derived-type-spec***)
		or	CLASS (*)
		or	TYPE (*)
		or	TYPEOF (*data-ref*)
		or	CLASSOF (*data-ref*)

R703a ***alter-type-spec*** **is** *type-spec-list*

Constraint: An *alter-type-spec* shall be one *type-spec* unless it appears in a generic type declaration statement.

R703b ***alter-derived-type-spec*** **is** *derived-type-spec-list*

Constraint: An *alter-derived-type-spec* shall be one *derived-type-spec* unless it appears in a generic type declaration statement..

R702(asis) *type-spec*	**is**	*intrinsic-type-spec*
	or	*derived-type-spec*
	or	*enum-type-spec*
	or	*enumeration-type-spec*

C703(asis) The *derived-type-spec* shall not specify an abstract type (F023:7.5.7).

NOTE 1

In the generic subprogram of NOTE1 of 3.1, the generic function ABSMAX has the generic type declaration statement:

```
    TYPE(INTEGER,REAL,DOUBLE PRECISION) :: X(:)
```

represents that the type of the argument `X` is one of default integer, default real, and double precision. Thereby, the generic subprogram produces specific procedures corresponding to the types, respectively.

NOTE2

The following is an example of a generic subprogram that provides two specific procedures, whose types of the arguments are 32-bit real and `mytyp1` with the type parameter `p1`.

```
    GENERIC SUBROUTINE swap(x,y)
      USE :: iso_fortran_env, ONLY: real32
      USE :: mymod, ONLY: mytyp1, p1
      TYPE(REAL(real32), mytyp1(p1)) :: x(:), y(:), tmp(:)

      tmp = x      !  Assignment(=) must be predefined for mytyp1(p1).
      x = y
      y = tmp
    END SUBROUTINE
```

Comment:

- TYPE(...) and CLASS(...) do not appear together in a *declaration-type-spec*. Therefore, both intrinsic and abstract types cannot be alternative types, and both non-abstract and abstract derived types cannot be alternative types. It might be relaxed if there were use cases.
- A type-generic subprogram can only unite specific subprograms that have exactly the same program code except for type declaration statements. To allow partially different program codes, one of the following extensions may be helpful.

– Use a new META SELECT TYPE construct; unlike the SELECT TYPE construct, the *selector* of the META SELECT TYPE construct shall be nonpolymorphic and the processor selects the one of constituent blocks at compile time.

```
GENERIC FUNCTION foo(x) RESULT(y)
  TYPE(type1,type2) :: x, y
    !! code if x is type1 or type2
  META SELECT TYPE (x)
  META TYPE IS (type1)
    !! code if x is type1
  META TYPE IS (type2)
    !! code if x is type2
  END META SELECT
    !! code if x is type1 or type2
END FUNCTION foo
```

– Allow the SELECT TYPE construct to have the same role as above. Namely, the *selector* in the SELECT TYPE statement is extended to have a nonpolymorphic type, and then select a constituent block at compile time.

3.2.2 Alternative kind specifier for intrinsic type

The *intrinsic-type-spec* has been extended to have alternative kind parameters for intrinsic types.

R794(asis)	*intrinsic-type-spec*	**is**	*integer-type-spec*
		or	REAL [*kind-selector*]
		or	DOUBLE PRECISION
		or	COMPLEX [*kind-selector*]
		or	CHARACTER [*char-selector*]
		or	LOGICAL [*kind-selector*]

R705(asis)	*integer-type-spec*	**is**	INTEGER [*kind-selector*]

Constraint: DOUBLE PRECISION and REAL with *kind-selector* shall not appear in the same *alter-type-spec*.

Constraint: If an *intrinsic-type-spec* without *kind-selector* appears in an *alter-type-spec*, other *intrinsic-type-spec*s of the same type shall not appear in the *alter-type-spec*.

The *kind-selector* and the *char-selector* are extended to have alternative kind parameters.

R706x	*kind-selector*	**is**	([KIND =] ***alter-kind-spec***)

R706a	***alter-kind-spec***	**is**	*
		or	*kind-spec-list*
R706b	*kind-spec*	**is**	*scalar-int-constant-expr*
R721x	*char-selector*	**is**	*length-selector*
		or	([LEN =] *type-param-value* , KIND = ***alter-kind-spec***)
		or	(KIND = ***alter-kind-spec*** [, LEN = *type-param-value*])

An *alter-kind-spec* designated as an asterisk specifies that the alternative kind parameters are all kind type parameters for the intrinsic type supported by the processor. An *alter-kind-spec* designated by *kind-spec-list* specifies that the alternative kind parameters are the values of *kind-spec-list*.

Constraint: In a generic type declaration statement, a *kind-spec* shall not have the same value as any other *kind-spec* in the same *intrinsic-type-spec* or in any *intrinsic-type-spec* that is of the same type.

Constraint: An *alter-kind-spec* shall be just one *kind-spec* except when it appears in the *intrinsic-type-spec* of a generic type declaration statement.

NOTE 1

In a generic type declaration statement:

```
TYPE(INTEGER(2,4)) :: X, Y
```

represents that either both X and Y are of integer(kind=2), or both X and Y are of integer(kind=4). The corresponding specific procedures are two. The statement can also be rewritten as follows, keeping the meaning:

```
TYPE(INTEGER(2,4)) :: X
TYPEOF(X) :: Y
```

Next, the following combination of type declaration statements:

```
TYPE(INTEGER(2,4)) :: X
TYPE(INTEGER(2,4)) :: Y
```

has a different meaning from the previous example. It represents four alternatives that correspond to four specific procedures, as follows:

```
TYPE(INTEGER(2)) :: X; TYPE(INTEGER(2)) :: Y
TYPE(INTEGER(4)) :: X; TYPE(INTEGER(2)) :: Y
TYPE(INTEGER(2)) :: X; TYPE(INTEGER(4)) :: Y
TYPE(INTEGER(4)) :: X; TYPE(INTEGER(4)) :: Y
```

NOTE 2

Examples of type declaration statements with alternative types and kinds are:

```
TYPE(INTEGER, LOGICAL) :: A
INTEGER(kind=2,4),DIMENSION(10,10) :: B
TYPE(INTEGER(kind=2,4), REAL(*), MYTYPE) :: X, Y(100)
```

Where `MYTYPE` is the name of a derived type. If the processor supports kind type parameters 4, 8, and 16 for real type, the last statement above represents the following set of alternative type declaration statements.

```
TYPE(INTEGER(kind=2)) :: X, Y(100)
TYPE(INTEGER(kind=4)) :: X, Y(100)
TYPE(REAL(kind=4)) :: X, Y(100)
TYPE(REAL(kind=8)) :: X, Y(100)
TYPE(REAL(kind=16)) :: X, Y(100)
TYPE(MYTYPE) :: X, Y(100)
```

3.2.3 Alternative kind specifier for parameterized derived type

The *derived-type-spec* has been extended to have alternative kind parameters for parameterized derived types.

R754(asis) *derived-type-spec* **is** *type-name* [(*type-param-spec-list*)]

C795(asis) *type-name* shall be the name of an accessible derived type.

C796(asis) *type-param-spec-list* shall appear only if the type is parameterized.

C797(asis) There shall be at most one *type-param-spec* corresponding to each parameter of the type. If a type parameter does not have a default value, there shall be a *type-param-spec* corresponding to that type parameter.

R755x *type-param-spec* **is** *type-param-value*
or *keyword* = ***alter-type-param-value***

Instead of C798: A *type-param-spec* shall not be a *type-param-value* unless all preceding *type-param-spec*s in the *type-param-spec-list* are *type-param-value*s.

C799(asis) Each *keyword* shall be the name of a parameter of the type.

NOTE 1

Syntactically, the *keyword* = acts as a separator between *type-param-spec*s in the list. That is, *type-param-spec*s are separated by a comma before the first appearance of the *keyword*, or by ", *keyword* =" thereafter.

R701(asis)	*type-param-value*	**is**	*scalar-int-expr*
		or	*
		or	:

C701(asis) The *type-param-value* for a kind type parameter shall be a constant expression.

C702(asis) A colon shall not be used as a *type-param-value* except in the declaration of an entity that has the POINTER or ALLOCATABLE attribute.

C7100(asis) An asterisk shall not be used as a *type-param-value* in a *type-param-spec* except in the declaration of a dummy argument or associate name or in the allocation of a dummy argument.

R701a	***alter-type-param-value***	is	*scalar-int-expr-list*
		or	*
		or	:

Instead of C701: An *alter-type-param-value* corresponding to a kind type parameter shall be a list of scalar integer constant expressions if it appears in a generic type declaration statement, or a scalar integer constant expression otherwise.

Constraint: An *alter-type-param-value* that does not correspond to a kind type parameter shall be a scalar integer expression, an asterisk, or a colon.

Constraint: Any two *scalar-int-expr*s in a *scalar-int-expr-list* shall not have the same value.

Constraint: If two or more *derived-type-spec*s with the same *type-name* appear in the *declaration-type-spec* of a generic type declaration statement, every two of the *derived-type-spec*s meet the following conditions. Here, for a kind parameter, alternative kind values are values of *scalar-int-expr*s if the *type-param-spec* is specified, or the default value otherwise.

– The derived type shall have at least one kind parameter.
– For at least one kind parameter of the derived type, there should be no overlap between each alternative kind values.

Same as C702: A colon shall not be used as a *type-param-value* except in the declaration of an entity that has the POINTER or ALLOCATABLE attribute.

Same as C7100: An asterisk shall not be used as a *type-param-value* in a *type-param-spec* except in the declaration of a dummy argument or associate name or in the allocation of a dummy argument.

NOTE 2

A dummy argument specified in the generic type declaration statement must be distinguishable (F2023: 15.4.3.4.5) among the specification procedures created. The constraints on parameterized derived types are intended to avoid this situation. The examples are shown below.

For the following type definition:

```
type mytyp(k, m, n)
   integer, kind :: k = 4
   integer, kind :: m
   integer, len :: n = 100

   real(k) :: a(m, n)
end type mytyp
```

the following *declaration-type-spec*s are correct in generic type declaration statements,

- `type( mytyp(8,100,100) )`
- `type( mytyp(k=8,m=100,200,n=50) )`
- `type( mytyp(m=10,20), mytyp(m=30) )`
- `type( mytyp(4,m=10,20), mytyp(8,m=20,30) )`
- `type( mytyp(m=10,20), mytyp(8,m=20,30) )`
- `type( mytyp(m=10,20,30,k=8), mytyp(m=20),mytyp(m=30,40) )`

and the following *declaration-type-spec*s are incorrect in generic type declaration statements.

- `type( mytyp(k=8,m=100,200,100,n=50) )`
  Error: the pair k=8 and m=100 appears twice.
- `type( mytyp(8,m=10,20), mytyp(8,m=20,30) )`
  Error: the pair k=8 and m=20 appears twice.
- `type( mytyp(m=10,20), mytyp(4,m=10,20) )`
  Error: the pair k=4 (default) and m=10 and the pair k=4 and m=20 appear twice.
- `type( mytyp(m=10,20,n=100), mytyp(m=10,40,n=200) )`
  Error: the pair k=4 (default) and m=10 appears twice. The LEN parameter `n` is not relevant for the distinction.

3.2.4 Alternative rank specifier

A type declaration statement has alternative ranks if the *rank-clause* as an *attr-spec* has two or more *rank-specs*.

The *rank-clause* is extended to have alternative ranks and to have the RANKOF keyword, as follows.

R829x	*rank-clause*	**is**	RANK (***rank-value-range-list***)
		or	**RANKOF (** ***data-ref*** **)**

Constraint: A *data-ref* shall not be *assumed-rank*.

R1148a ***rank-value-range*** **is** *rank-value*
or *rank-value* :
or : *rank-value*
or *rank-value* : *rank-value*

R1149a *rank-value* **is** *scalar-int-constant-expr*

Constraint: A *rank-value* in *rank-value-range-list* shall be nonnegative and the value is less than or equal to the maximum rank supported by the processor.

The interpretation of *rank-value-range-list* is the same as the one of *case-value-range-list* described in F2023:11.1.9.2 "Execution of a SELECT CASE construct". The alternative ranks specified in *rank-clause* are all ranks for which matching occurs.

Constraint: A *rank-value-range* shall be just one *rank-value* except in a *rank-clause* of a *type-declaration-statement* appearing in the specification part of a generic subprogram.

RANKOF with a *data-ref* specifies the same rank as the declared rank of *data-ref*.

NOTE 1

Examples of type declaration statements with alternative ranks are:

```
REAL(8), RANK(0:3) :: A
TYPE(REAL(8)), RANK(1,2,3) :: B
REAL, RANK(10:) :: X, Y(100)
```

If the maximum array rank supported by the processor is 15, the last statement above represents the following alternative TYPE declaration statements.

```
REAL, RANK(10) :: X, Y(100)
REAL, RANK(11) :: X, Y(100)
REAL, RANK(12) :: X, Y(100)
REAL, RANK(13) :: X, Y(100)
REAL, RANK(14) :: X, Y(100)
REAL, RANK(15) :: X, Y(100)
```

Comment:

- The RANK clause cannot specify lower and upper bounds of assumed-shape arrays. So further extension might be allowed, for example:
  - `REAL(8), DIMENSION(0:),(:, 2:10),(0:,:,:) :: A`
  - `REAL(8) :: A(0:),(:, 2:10),(0:,:,:)`

- A rank-generic subprogram can only unite specific subprograms that have exactly the same program code except for type declaration statements. To allow partially different program codes, one of the following extensions may be helpful.
  - Use a new META SELECT RANK construct; unlike the SELECT RANK construct, the *selector* of the META SELECT RANK construct shall not be assumed-rank and the processor selects the one of constituent blocks at compile time. The program of List 2 in 2.2 can be written using the construct as follows.

```
GENERIC FUNCTION has_nan(x) RESULT(ans)
  REAL(REAL32,REAL64,REAL128), RANK(0:15), INTENT(IN) :: x
  LOGICAL :: ans
  META SELECT RANK (x)
  META RANK (0)
    ans = ieee_is_nan(x)
  META RANK (1:15)
    ans = any(ieee_is_nan(x))
  END META SELECT
END FUNCTION has_nan
```

  - Allow the SELECT RANK construct to have the same role as above. Namely, the *selector* in the SELECT TYPE statement is extended to be able to have a non-assumed-rank variable name, and then select a constituent block at compile time.

4. Summary

This paper proposed the following language extensions for the generic subprogram:

- The GENERIC prefix and the *generic-spec* allowed in a FUNCTION and SUBROUTINE statements,
- Listed type specifiers of the *declaration-type-spec* in a type declaration statement,
- Listed kind specifiers and * of the *kind-selector* or *char-selector* in a type declaration statement,
- *rank-value-range-list* of RANK clause, and
- RANKOF clause in a type declaration statement.

A function or subroutine subprogram with the GENERIC prefix is a generic subprogram. A generic subprogram defines multiple specific procedures and the generic identifier, that is a generic name, an operator or assignment, or a defined I/O.

So far, the generic names, operators, and defined I/O provided by the generic identifier mechanism bring great convenience to library users. However, this often required the library provider to create tens or hundreds of specific subprograms; otherwise, they had no choice but to program in a processor-dependent manner or to program leaving decision and branch costs at runtime. Since the generic subprogram significantly reduces the size of the code that describes the specific subprograms, it reduces programming and maintenance costs without compromising execution performance and portability.

5. Acknowledgments

I would like to thank John Reid for reading this paper and suggesting some improvements to the presentation and Tomohiro Degawa and the user group Fortran-jp for discussing it from the user's perspective and offering practical suggestions. And I also thank Hiroyuki Sato for useful comments, and Masayuki Takata and Toshihiro Suzuki for pointing out improvements in examples and descriptions.

History

Version 1.0 → 1.1

- Multiple type specs are allowed not only in the TYPE clause but also in the CLASS clause.
  - R703x was modified with CLASS (alter-derived-type-spec).
  - R703b was added.
  - Three constraints are added:
    - Constraint: Any two type-specs in a type-spec-list shall not be the same type specifier.
    - Constraint: An alter-derived-type-spec shall be just one derived-type-spec except in a declaration-type-spec of a type-declaration-statement appearing in the specification part of a generic subprogram.
    - Constraint: Any two derived-type-specs in a derived-type-spec-list shall not be the same type specifier.
  - Comments about the difference between TYPE and CLASS clauses were eliminated.
- Comment about the idea of TYPE(INTRINSIC), TYPE(ARITHMETIC), etc. were eliminated.
- Mentioned the META SELECT TYPE construct in Comments of 3.2.1.
- Mentioned the META SELECT RANK construct in Comments of 3.2.3.

Version 1.1 → 1.2

- The title was changed from "Generic Subprogram" to "Proposal for Generic Subprogram."

Version 1.2 → 1.3

- In List 1, "LOC" was replaced by "lines of code" in three places.
- In 3.1 and 4, "function or subroutine statement" to "FUNCTION or SUBROUTINE statement."
- In NOTE 3 of 3.1, modified from:

  ```
  TYPE(MYTYPE1, MYTYPE2),INTENT(IN) :: X,Y
  ```

  to:

  ```
  TYPE(MYTYPE1, MYTYPE2),INTENT(IN) :: X
  TYPE(MYTYPE1, MYTYPE2),INTENT(IN) :: Y
  ```
- In Comment of 3.2.1, added more explanations and one alternative idea.
- In the second item of Comment of 3.2.3, added more explanations and one alternative idea.
- In 5 Acknowledgment, added thanks to Schuko, Makki, and Suzu-P.
- Some typos and trivial modifications.

Version 1.3 → 1.4

- In 1. Introduction, improved.
- In 3. Syntax, a new term generic type declaration statement is defined and used.
- In 3, *generic-spec* is allowed in *function/subroutine-stmt* instead of *function/subroutine-name.*
- In 3.1, add NOTE 3 and 4.
- In 3.2.1, add NOTE 1 and 2.
- Add "3.2.3 Alternative kind specifier for parameterized derived type" and reorganized in 3.2.
- In 4. Summary, modified.