Proposal for Generic Subprogram

Version 1.3
Hidetoshi Iwashita
June 8, 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 orders or argument types, kinds, or ranks differ from each other. In Fortran, most intrinsic procedures and operators are generic, e.g., 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 for user-defined (derived) types as well.

Importantly, using a generic identifier does not reduce execution performance as compared to directly using the specific name selected. In programming languages in general, there is often a trade-off between abstraction and execution performance. However, in Fortran, there is no performance degradation in the generic identifier because the following points are considered:

- Selecting a specific procedure depends only on static information 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 advanced optimization and code generation within the compiler back-end.

A major challenge for the generic identifier mechanism is the 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

  !!--- real32, rank0-15
  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

  !!--- real64, rank0-15
  ... (omit 80 lines of code)
  !!--- real128, rank0-15
  ... (omit 75 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** 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 subprogram is a subprogram that has the GENERIC prefix (3.1). A type declaration statement in a generic subprogram is extended to specify alternative types (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* (F2023:15.6.2.1) is extended as follows.

R1530x *prefix-spec* **is** …
or GENERIC

Constraint: If the GENERIC prefix appears in an *interface-body*, the containing *interface-block* shall be a generic interface block. If the interface block has a *generic-name*, all *function-name*s or *subroutine-name*s of *interface-bodie*s with the GENERIC prefix shall be the same as the *generic-name* of the *interface-block.*

The description of F2023:15.6.2.2 (Function subprogram) paragraph 2 is changed as follows:

OLD: The name of the function is *function-name*.

NEW: If GENERIC does not appear in the prefix, the name of the function is *function-name*. If GENERIC appears in the prefix, the name of the generic function is *function-name* and the name of each specific function procedure is undefined.

The description of F2023:15.6.2.3 (Subroutine subprogram) paragraph 2 is changed as follows:

OLD: The name of the subroutine is *subroutine-name.*

NEW: If GENERIC does not appear in the prefix, the name of the subroutine is *subroutine-name.* If GENERIC appears in the prefix, the name of the generic subroutine is *subroutine-name* and the name of each specific subroutine procedure is undefined.

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 when the module generic function ABSMAX of NOTE 1 is 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

Using the interface block, a generic subprogram can be defined as an operator, assignment, or *defined-io-generic-spec* (Fortran2023:R1509). For example, the following interface block defines that "*A*+*B*" means "MYADD(*A*,*B*)", where the type of *A* is MYTYPE1 or MYTYPE2, the type of *B* is MYTYPE1 or MYTYPE2, and the type of the result is the same as the type of *A*.

```
INTERFACE OPERATOR(+)
   GENERIC FUNCTION MYADD(X,Y) RESULT(Z)
     TYPE(MYTYPE1, MYTYPE2),INTENT(IN) :: X
     TYPE(MYTYPE1, MYTYPE2),INTENT(IN) :: Y
     TYPEOF(X) :: Z
   END FUNCTION MYADD
END INTERFACE
```

Comment:

- Constraints for the interface block seems not sufficient.
- 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(as is) *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), and ranks (3.2.3).

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 the *array-spec*.

3.2.1 Alternative type specifier

A type declaration statement has alternative types if the *declaration-type-spec* has two or more *type-specs*.

The *declaration-type-spec* is 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 just one *type-spec* except in a *declaration-type-spec* of a *type-declaration-statement* appearing in the specification part of a generic subprogram.

Constraint: Both DOUBLE PRECISION and REAL with *kind-selector* shall not appear in a *type-spec-list*.

Constraint: Any two *type-spec*s in a *type-spec-list* shall not be the same type specifier.

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

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-spec*s in a *derived-type-spec-list* shall not be the same type specifier.

NOTE 1

A *data-component-def-stmt* (F2023:R737), the prefix of a *function-stmt* (F2023:R1529), or an *implicit-spec* (F2023:R867) cannot specify alternative types for entities in the *entity-decl-list*.

NOTE 2

In a generic subprogram, 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
```

Comment:

- 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

A type declaration statement has alternative kinds if a *kind-selector* or a *char-selector* in the *declaration-type-spec* has two or more *kind-spec*s.

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 * 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: Any two kind parameters specified in an *alter-kind-spec* shall not have the same value.

Constraint: An *alter-kind-spec* shall be just one *kind-spec* except in a *declaration-type-spec* of a *type-declaration-statement* appearing in the specification part of a generic subprogram.

NOTE 1

A *data-component-def-stmt* (F2023:R737), the prefix of a *function-stmt* (F2023:R1529), or an *implicit-spec* (F2023:R867) cannot specify alternative kind parameters for entities in the *entity-decl-list*.

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 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)
```

Comment:

- Derived types might also have alternative kind parameters.

3.2.3 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 of a FUNCTION or SUBROUTINE statement,
- Listed type specifiers of the *declaration-type-spec* in a type declaration statement,
- Listed kind specifiers or * of the *kind-selector* or *char-selector* in a type declaration statement, and
- *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 the generic name and multiple specific procedures with no names. The generic name can also be associated with an operator or assignment.

The generic names and operators 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.