総称副プログラムの提案
1.7J 版

岩下 英俊
情報処理学会情報規格調査会 Fortran WG 小委員会
2023 年 12 月 23 日

1. はじめに

総称識別子による個別手続選択機構は、Fortran の優れた特徴である。総称識別子（総称名、利用者定義の演算子若しくは=、または利用者定義入出力）は、引数の型、種別、または次元数が互いに異なる個別手続の中から一つを識別する。Fortran では、ほとんどの組込み手続と演算子が“総称”である。例えば、組込み関数 MAX の引数は整数型、実数型、または文字型であり、演算子 + のオペランドは整数型、実数型、または複素数型である。総称名や総称演算子を使用するのは、自然で生産的なプログラミング・スタイルである。ユーザ定義手続や派生型についても同様である。

重要なのは、総称識別子を使っても実行性能に影響を与えないことである。性能を犠牲にしないことは、Fortran において不可欠な要件である。総称識別子のメカニズムは、次のような配慮によって実現されている。

- 個別手続の選択は静的なパラメタのみに依存し、コンパイル時に決定される。したがって、実行コードには判定や分岐のオーバーヘッドは残らない。
- 総称識別子は、コンパイラのフロントエンド中、あるいはその前に解決されるため、コンパイラのバックエンド内の既存の高度な最適化とコード生成には影響を与えない。

このように、総称識別子のメカニズムは、ユーザにとって利便性と性能を両立させる機能であると言える。一方、個別手続を作成して総称識別子として公開するライブラリ提供者にとっては、依然として組合せ爆発という大きな課題を抱えている。プログラマがライブラリ手続の型や次元数を一般化しようとすると、個別副プログラムは数十から数百という膨大な数に膨れ上がる可能性がある。例えば、引数変数が任意の算術型（整数、実数、または複素数型であり、種別は任意）と任意の次元数（標準では 0 から 15）を持つ関数を定義するためには、プログラマは 100 以上の個別関数副プログラムを書かなければならない。そのような膨大な数の個別副プログラムが賢いエディターやツールを使って書けたとしても、そのような数のバージョンを保守・改良するのは、エラーが起こりやすく時間の無駄である。

本稿は、総称識別子機構を拡張し、大量の個別手続を簡単に定義することを提案する。ユーザは多数の個別副プログラムを記述する代わりに、複数の個別手続を定義する**総称副プログラム**を記述すればよい。

本稿では、2 章はまず手っ取り早く理解するための例を示す。3 章で文法を説明し、4 章でまとめる。

2. 例

引数が NaN（not a number）か、少なくとも一つの NaN 配列要素を持つ場合に真を返し、そうでない場合に偽を返す単純な関数を考える。引数は 32、64、128 バイトの実数型で、スカラまたは 1〜15 次元数の配列が許される。

2.1 一組の個別関数による記述

リスト 1 は、総称関数 has_nan と、全型、全次元数に対する 48 の個別関数を定義した例を示している。おわかりのように、ほとんどの関数の本体は同じだが、異なる次元数または異なる種別パラメタを持つため、現在の Fortran 標準では別々の関数として記述しなければならない。

2.2 総称副プログラムによる記述

リスト 2 は、リスト 1 のコードと等価なコードを、本稿で提案する総称副プログラムを使って記述したものである。GENERIC という接頭辞を持つ副プログラムは総称副プログラムである。最初の総称副プログラムは、三つの個別手続を定義しており、x は 32、64、128 バイトの実数型の一つである。二番目の総称副プログラムは、3×15 の個別手続を定義していて、x は 32、64、128 バイトの実数型と 1 から 15 までの次元数の組み合わせの一つである。総称副プログラムによって定義された個別手続はすべて名前を持たず、総称名によって参照される。

仮引数の総称型宣言文以外は同じ本体を持つ複数の個別副プログラムを、一つの総称副プログラムにまとめることができる。これにより、プログラムコード量を大幅に削減することができる。また、総称副プログラムは対応する個別手続の並びに展開されるため、性能の低下はないはずである。

リスト 1. 個別副プログラムで定義された has_nan

```
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 = iee_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

   ... (65 行のコードを省略)

   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

   ... (155 行のコードを省略)

   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
```

リスト 2. 総称副プログラムで定義された has_nan

```
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 = iee_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
```

3. 構文

総称副プログラムとは、GENERIC 接頭辞 (3.1) を持つサブプログラムで、互いに異なる型、種類、次元数の仮引数を持つ一つ以上の個別手続を定義する。総称副プログラム名は、定義されたすべての個別手続の総称名である。各個別手続は特定の名前を持たない。

総称型宣言文は、総称副プログラムの宣言部に現れる、代替する型、種別または次元数を指定する型宣言文である (3.2)。

引用仕様宣言（interface block）と GENERIC 文は、総称副プログラムの明示的引用仕様を指定し、総称副プログラムと総称識別子（利用者定義演算子、利用者定義代入、利用者定義入出力を含む）を関連付けるために拡張される (3.3

SELECT RANK 構文と SELECT TYPE 構文が拡張され、代替型、代替種別及び代替次元数に基づいたコードの切り替えが可能になった (33.4。

3.1 GENERIC 接頭辞

FUNCTION 文または SUBROUTINE 文の GENERIC 接頭辞は、その副プログラムが総称副プログラムであることを指定する。

prefix-spec、*function-stmt* 及び *subroutine-stmt* (F2023:15.6.2.1-3)は以下のように拡張される。

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***

制約: *function-spec* は、接頭辞に GENERIC が現れる場合は *generic-spec* とし、そうでない場合は関数名（*function-name*）とする。

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

制約:接頭辞に GENERIC が現れる場合、*proc-language-binding*-spec は現れてはならない。

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

制約: *subroutine-spec* は、接頭辞に GENERIC がある場合は *generic-spec* とし、そうでない場合はサブルーチン名（*subroutine-name*）とする。

代替案 1

R1553a、R1538a、および二つの制約のすべての ***generic-spec*** を ***generic-name*** に変更する。すなわち、OPERATOR(*defined-operator*)、ASSIGNMENT(=)、及び *defined-i/o-generic-spec* は、*function-stmt* の *function-spec* または subroutine-*stmt* の *subroutine-spec* として許されないとする。

これは機能的な低下ではないことに注意されたい。GENERIC 手続を任意の *generic-spec* に関連付けるには、プログラマは 3.3 に示すように総称引用仕様宣言を使用することができる。

注記 1

以下は、総称副プログラムをモジュール副プログラムとして持つモジュールの例である。

```
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
```

ここで TYPE(INTEGER,REAL,DOUBLE PRECISION)は、X が個別手続によって整数型、実数型、または倍精度型であることを指定する(0)。二つのモジュール副プログラムは総称副プログラムであり、同じ総称名を指定する。それらの引用仕様は明示的であるため、親スコープと兄弟スコープで参照することができる。従って、上記のプログラムは次のプログラムと等価である。

```
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
```

注記 2

総称副プログラムは外部副プログラムとすることができる。注記 1 の二つのモジュール総称関数が仮に外部関数だった場合、ABSMAX の引用仕様宣言を次に示す。

```
      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
```

注記 3

演算子 + の機能を拡張する総称副プログラムを例示する。

```
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
```

coord_t 型は、共通の種別がパラメタ化された実数型の構成要素 x、y 及び z を持つ。総称副プログラムは、coord_t(real32)型のオブジェクト間の二項演算 + と、coord_t(real64)型のオブジェクト間の二項演算 + を定義する。

総称サブプログラムに対する利用者定義演算子は、FUNCTION 文や SUBROUTINE 文だけではなく、総称引用仕様宣言または GENERIC 文でも定義することができる。次のプログラムは上記のプログラムと等価である。

```
MODULE coord_m
  USE iso_fortran_env

  TYPE coord_t(k)
     INTEGER, KIND :: k
     REAL(kind=k) :: x, y, z
  END TYPE coord_t
  PRIVATE :: coord_add
  GENERIC :: OPERATOR(+) => coord_add

CONTAINS
  GENERIC FUNCTION coord_add 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 coord_add
```

```
    END MODULE coord_m
```

なお、代替案 1 においては、前者の書き方を禁止し、後者の書き方だけを許している。

注記 4

次の例は、利用者定義入出力を定義する総称副プログラムである。ここで、coord_t は注記 3 のモジュール coord_m の中で定義されている。

```
  GENERIC SUBROUTINE WRITE(FORMATTED)(data, unit, iotype, v_list, iostat, iomsg)
    use coord_m
    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)
```

総称副プログラムは、coord_t(real32)型と coord_t(real64)型に対する書式付き出力文の DT 編集記述子の動作を定義している。この総称副プログラムを使用すると、次のコードが動作する:

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

    cod32%x = 1.11111111111111111d0
    cod32%y = 2.22222222222222222d0
    cod32%z = 3.33333333333333333d0
    write(*, "(DT(20,17))") cod32

    cod64%x = 1.11111111111111111d0
    cod64%y = 2.22222222222222222d0
    cod64%z = 3.33333333333333333d0
    write(*, "(DT(20,17))") cod64
```

以下に結果の例を示す:

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

コメント

- 個別手続名が未定義である。個別手続を名前あるいは他の方法で識別する必要があるのだろうか？もしそうなら、どのように指定できるのか？
  - 実引数は手続名でもよく、これは個別手続名でなければならない。「第 1 引数がデフォルトの実数型の場合の ABSMAX」というような表記が必要だろうか。
  - C 言語から総称手続を呼び出す必要があるようだ。この場合、BIND 文を拡張する必要があるのだろうか？例えば

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

3.2 総称型宣言文

総称型宣言文 (generic type declaration statement) は、代替する型、種別または次元数を指定する型宣言文である。型宣言文 (*type-declaration-stmt*) は、Fortran 2023 では次のように定義される。

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

declaration-type-spec は、代替する型 (0) または種別 (3.2.2, 3.2.3) を指定するために拡張される。*attr-spec* は、代替する次元数 (3.2.4) を指定するために拡張される。

制約: *declaration-type-spec* が代替する型または種別をもつとき, *entity-decl-list* の少なくとも一つの実体は, 仮引数でなければならない。

制約: *attr-spec* が代替する次元数をもつとき、*entity-decl-list* の少なくとも一つの実体は、*array-spec* をもたない仮引数でなければならない。

注記 1

データ成分定義文 (*data-component-def-stmt,* F2023:R737)、FUNCTION 文 (*function-stmt,* F2023:R1529) の手続接頭句 (prefix)、または暗黙型宣言(*implicit-spec*, F2023:R867) は、代替する型または種別を *entity-decl-list* 中にもたない。

3.2.1 型指定子の拡張

declaration-type-spec を拡張し、代替する型を指定できるようにする。

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

制約: *alter-type-spec* は、総称型宣言文中に現れない限り、一つの *type-spec* でなければならない。

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

制約: *alter-derived-type-spec* は、総称型宣言文中に現れない限り、一つの *derived-type-spec* でなければならない。

注記 1

3.1 の注記 1 の最初の総称副プログラムにおいて、総称関数 ABSMAX は次の総称型宣言文をもつ。

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

この文は引数 X が基本整数型、基本実数型、基本倍精度型のいずれかであることを表す。これにより、総称副プログラムはそれぞれの型に対応する個別手続を生成する。

注記 2

以下は、二つの個別手続を提供する総称副プログラムの例である。それらの引数の型は、それぞれ 32 ビット実数型、及び、型パラメタ p1 をもつ mytyp1 型である。

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

  tmp = x
  x = y
  y = tmp
END SUBROUTINE
```

コメント

- 代替の仮引数は、互いに区別できなければならない（F2023:15.4.3.4.5）。このルールにはいくつかの制約を加えるべきかもしれない。
- TYPE(...) と CLASS(...) は、一つの *declaration-type-spec* の中に一緒に現れることはない。したがって、組込み型と抽象型との両方を代替型にすることはできず、非抽象派生型と抽象派生型の両方を代替型にすることはできない。ユースケースがあれば緩和してもよいが。

3.2.2 組込み型の種別指定子の拡張

組込み型指定子 (*intrinsic-type-spec*) を拡張し、組込み型に代替する種別型パラメタを指定できるようにする。

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*]

制約：DOUBLE PRECISION と、*kind-selector* をもつ REAL とを、同じ *alter-type-spec* に書いてはならない。

制約：*kind-selector* を持たない *intrinsic-type-spec* が *alter-type-spec* に現れる場合、同じ型の他の*組込み型-spec* は*分身型-spec* に現れてはならない。

kind-selector と *char-selector* は、代替する種別パラメタを持つように拡張される。

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 (*type-param-value* , *kind-spec*)

or ([LEN =] *type-param-value* , KIND = ***alter-kind-spec***)

or (KIND = ***alter-kind-spec*** [, LEN = *type-param-value*])

アスタリスクで指定された *alter-kind-spec* は、代替する種別パラメタが、プロセッサがサポートする組込み型のすべての種別型パラメタであることを指定する。*kind-spec-list* で指定された *alter-kind-spec* は、代替する種別パラメタが、*kind-spec-list* の値であることを指定する。

制約: 総称型宣言文において、*kind-spec* は、同じ *intrinsic-type-spec* 内の他の *kind-spec*、または、同じ型の他の *intrinsic-type-spec* 内の *kind-spec* と、同じ値であってはならない。

制約: *alter-kind-spec* は、総称型宣言文の *intrinsic-type-spec* に現れる場合を除き、一つの *kind-spec* でなければならない。

注記 1

総称型宣言文の中で、

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

は、X と Y との両方が integer(kind=2)であるか、X と Y との両方が integer(kind=4)であることを表す。対応する個別手続は二つである。この文は、意味を保ったまま次のように書き換えることもできる。

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

次に、総称型宣言文の組み合わせ

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

は、前の例とは意味が異なる。これは次のように、四つの個別手続に対応する四つの選択肢を表している。

```
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
```

注記 2

代替する型と種別を持つ総称型宣言文の例は以下の通りである:

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

MYTYPE は派生型の名前である。プロセッサが実数型の種別パラメタ 4、8、16 をサポートしている場合、上記の最後の文は、以下の代替型宣言文のセットを表す。

```
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 派生型の種別指定子の拡張

派生型指定子 (*derived-type-spec*) を拡張し、パラメタ化された派生型に、代替する種別型パラメタを指定できるようにする。

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

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

C798 の変更： *type-param-spec-list* 内の *type-param-spec* は、先行するすべての *type-param-spec* が *type-param-value* でない限り， *type-param-value* であってはならない。

注記 1

構文上、*keyword* = は *type-param-spec-list* 中の区切り文字として機能する。つまり、*type-param-spec* は、*keyword* が最初に現れる前ではカンマで区切られ、それ以降は ", *keyword* =" で区切られる。

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

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

制約： 種別型パラメタに対応する *alter-type-param-value* は、総称型宣言文に現れる場合はスカラ整数定数式の並びでなければならず、そうでない場合はスカラ整数定数式でなければならない。

制約: 種別型パラメタに対応しない *alter-type-param-value* は、スカラ整数式、アスタリスク、またはコロンでなければならない。

制約： *scalar-int-expr-list* 中の二つの *scalar-int-expr* は、同じ値をもってはならない。

制約: 総称型宣言文の *declaration-type-spec* 中に同じ型名 (*type-name*) をもつ二つ以上の *derived-type-spec* が現れる場合、任意の二つの *derived-type-spec* は以下の条件を満たさなければならない。ここで、種別パラメタについて、*type-param-spec* が指定されていれば代替する種別値は *scalar-int-expr* の値であり、そうでなければ暗黙の値である。

- 派生型は少なくとも一つの種別パラメタを持たなければならない。
- 派生型の少なくとも一つの種別パラメタについては、それぞれの代替する種別値の間に重複があってはならない。

C702 と同じ制約： コロンは，POINTER 属性または ALLOCATABLE 属性を持つ実体の宣言以外では，*alter-type-param-value* として使用してはならない。

C7100 と同じ制約： アスタリスクは、仮引数若しくは関連名の宣言、または仮引数の割り当てを除き、*alter-type-param-value* として使用してはならない。

注記 2

総称型宣言文に指定された仮引数は、作成される個別手続の間で区別可能(F2023: 15.4.3.4.5)でなければならない。パラメタ化された派生型に対する制約は、このような状況を避けるためのものである。以下にその例を示す。

以下の型定義について:

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

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

総称型宣言文内では、以下の *declaration-type-spec* は正しい。

- `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) )`

以下の *declaration-type-spec* は正しくない。

- `type( mytyp(k=8,m=100,200,100,n=50) )`
  Error: k=8 と m=100 の組合せが 2 度出現
- `type( mytyp(8,m=10,20), mytyp(8,m=20,30) )`
  Error: k=8 と m=20 の組合せが 2 度出現
- `type( mytyp(m=10,20), mytyp(4,m=10,20) )`
  Error: k=4（デフォルト）と m=10 の組合せが 2 度出現。k=4 と m=20 の組合せが 2 度出現
- `type( mytyp(m=10,20,n=100), mytyp(m=10,40,n=200) )`
  Error: k=4（デフォルト）と m=10 の組合せが 2 度出現。なお、長さパラメタ n は個別手続の区別には関係しない。

3.2.4 次元数指定子の拡張

RANK 句 (*rank-clause*) を拡張し、代替する次元数を指定できるようにする。

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

制約: *data-ref* は次元数引継ぎ実体であってはならない。

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*

制約: *rank-value-range-list* の *rank-value* は、0 以上で、プロセッサがサポートする最大次元数以下でなければならない。

rank-value-range-list の解釈は、F2023:11.1.9.2 “SELECT CASE 構文の実行” に記述されている *case-value-range-list* の解釈と同じである。RANK 句で指定される代替する次元数は、マッチングするすべての次元数である。

制約: *rank-value-range* は、総称副プログラムの宣言部に現れる総称型宣言文の RANK 句を除き、一つの *rank-value* だけでなければならない。

RANKOF (*data-ref*) は RANK (RANK (*data-ref*)) と等価である。

注記 1

代替する次元数を持つ型宣言文の例は次の通りである:

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

プロセッサがサポートする配列の最大次元数が 15 の場合、上記の最後の文は、以下の代替する型宣言文を表す。

```
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)
```

コメント

- RANK 句では、次元数引継ぎ配列の下限と上限を指定することはできない。そのため、次の例のような拡張が必要かもしれない:
  - `REAL(8), DIMENSION(0:),(:, 2:10),(0:,:,:) :: A`
  - `REAL(8) :: A(0:),(:, 2:10),(0:,:,:)`

- RANKOF(data-ref) は常に RANK(RANK(data-ref)) に書き換えられるので、必ずしも必要ではない。しかし、次のフレーズは非常に便利である。

  - `TYPEOF(A), RANKOF(A) :: B`

3.3 引用仕様宣言の拡張

引用仕様宣言の**総称引用仕様本体**とは、その FUNCTION 文または SUBROUTINE 文に GENERIC 属性をもつ引用仕様本体である。総称引用仕様本体は、総称副プログラムの明示的な引用仕様を指定する。個別引用仕様宣言と総称引用仕様宣言はどちらも総称引用仕様本体を持つことができる。

制約: 総称引用仕様宣言の名前は、それが含むすべての総称引用仕様本体の名前と異なっていなければならない。

引用仕様宣言中の PROCEDURE 文 (*procedure-stmt*)、及び、GENERIC 文 (*generic-stmt*) は、次のように拡張される。

R1506x *procedure-stmt*	**is**	[MODULE] PROCEDURE [::] ***specific-spec-list***
R1507x ***specific-spec***	**is** **or**	*procedure-name* ***generic-spec***
R1510x *generic-stmt*	**is**	GENERIC [, *access-spec*] :: *generic-spec* => ***specific-spec-list***

F2023:15.4.3.2.1（総称識別子）の段落 2 にある次の文

interface-stmt 中の *generic-spec* は、引用仕様宣言中のすべての手続の総称識別子である。

を、次のように変更する。

interface-stmt 中の *generic-spec* は、引用仕様宣言中のすべての末端の個別手続の総称識別子である。

また、段落 3 にある次の文

総称名は、総称引用仕様内の手続名のどれか一つと同じであってもよいし、・・・

を、次のように変更する。

総称名は、総称引用仕様内の**個別**手続名のどれか一つと同じであってもよいし、・・・

F2023:15.4.3.3（GENERIC 文）の段落 1 にある次の文

GENERIC 文は、一つ以上の個別手続に対する総称識別子を指定し、・・・

を、次のように変更する。

GENERIC 名は、一つ以上の個別手続**または総称手続**に対する総称識別子を指定し、・・・

注記 1

次の個別引用仕様宣言は、整数または実数の引数を持つことができる総称手続 foo と、実数または複素数の引数を持つことができる総称手続 goo を指定している。

```
interface
  generic subroutine foo(a)
    type(integer, real) :: a
  end subroutine foo
  generic subroutine goo(a)
    type(real, complex) :: a
  end subroutine goo
end interface
```

上の例では、INTERFACE 文では *generic-spec* を指定できない。引数が同じ実数型のとき、foo と goo が区別できないからである。

以下の総称引用仕様宣言は、整数または実数の引数を持つことができる fooと、整数、実数または複素数の引数を持つことができる foobarという二つの総称手続を指定している。

```
interface foobar
  generic subroutine foo(a)
    type(integer, real) :: a
  end subroutine foo
  subroutine bar(a)
    type(complex) :: a
  end subroutine bar
end interface bal
```

注記 2

次のモジュールは、3.1 の注記 3 のモジュールと同等である。

```
MODULE coord_m
  USE iso_fortran_env

  PRIVATE :: add_coord

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

  INTERFACE OPERATOR(+)
    MODULE PROCEDURE :: add_coord
  END INTERFACE

CONTAINS

  GENERIC FUNCTION add_coord(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 add_coord

END MODULE coord_m
```

引用仕様宣言を使う代わりに、以下の GENERIC 文を使うこともできる。

```
  GENERIC ::OPERATOR(+) => add_coord
```

3.4 SELECT 構文の拡張

3.4.1 SELECT RANK 構文の拡張

SELECT RANK 構文を拡張し、選択子 (*selector*) として次元数引継ぎでない変数を指定できるようにする。

C1155x SELECT RANK 文 (*select-rank-stmt*) 中の選択子は、次元数引継ぎ配列の名前**または非次元数引継ぎ実体の名前**でなければならない。

制約: 選択子が次元数引継ぎでない場合、SELECT RANK 文中に *associate-name* を書くことはできない。

R1152x *select-rank-case-stmt*	**is**	RANK (***rank-value-range-list***) [*select-construct-name*]
	or	**RANKOF (*data-ref*) [*select-construct-name*]**
	or	RANK (*) [*select-construct-name*]
	or	RANK DEFAULT [*select-construct-name*]

rank-value-range は 3.2.4 の R1148a で定義されている。

制約: 選択子が次元数引継ぎである場合、*rank-value-range-list* は単一の次元数でなければならない。

制約: 選択子が次元数引継ぎである場合、RANKOF *select-rank-case-stmt* は許されない。

注記 1

SELECT RANK 構文は、選択子が次元数引継ぎであれば実行時に、そうでなければコンパイル時に、最大一つのブロックを選択する。総称副プログラムでは、プログラマは選択子として代替する次元数を持つ仮引数を指定することで、SELECT RANK 構文で次元数ごとに部分的に異なるプログラムコードを書くことができる。 例えば、2.2 のリスト 2 の二つの総称副プログラムは、SELECT RANK 構文を使って次のように書くことができる。

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

SELECT RANK 構文による総称副プログラムの統一は、コードの大部分は同じだが、わずかな部分だけが次元数によって異なる場合に便利である。

3.4.2 SELECT TYPE 構文の拡張

SELECT TYPE 構文を拡張し、選択子として多相的変数でない変数をサポートする。

F2023:11.1.11.1 の次の文が変更された。

選択は式の実行時の型に基づいて行われる。

にある:

選択は、選択子が多相的である場合は式の実行時の型に基づき、そうでない場合は式の宣言時の型に基づいて行われる。

C1164x (R1155) *select-type-stmt* の選択子は、多相的な言語要素、**または、非多相的なデータ実体**でなければならない。

R1156x *type-guard-stmt*	**is**	TYPE IS (***type-spec-list***) [*select-construct-name*]
	or	CLASS IS (***derived-type-spec-list***) [*select-construct-name*]
	or	**TYPEOF (*data-ref*) [*select-construct-name*]**
	or	**CLASSOF (*data-ref*) [*select-construct-name*]**
	or	CLASS DEFAULT [*select-construct-name*]

TYPEOF (*data-ref*) と CLASSOF (*data-ref*) は、それぞれ TYPE IS (*type-spec*) と CLASSOF (*type-spec*) と等価である。ここで、*type-spec* は、*data-ref* の型及び種別とする。

制約: 選択子が多相的である場合、*type-spec-list* は単一の *type-spec* でなければならず、*derived-type-spec-list* は単一の *derived-type-spec* でなければならない。

制約: 選択子が多相的である場合、TYPEOF 文や CLASSOF 文は許されない。

制約: 選択子が多相的でない場合、TYPE IS 文、CLASS IS 文、TYPEOF 文または CLASSOF 文で指定された *type-spec*、*derived-type-spec*、*data-ref* の種別型パラメタはすべて定数でなければならない。

C1167x (R1154) 選択子が**多相的であるが**無制限多相ではない場合、TYPE IS 文または CLASS IS 文は、それぞれ選択子の宣言された型の拡張を指定しなければならない。

コメント

- 長さ型パラメタの説明が不十分である。制約 C1165 は長さ型パラメタを仮定しなければならないと述べているが、総称副プログラムの仮引数には適さないかもしれない。
- 総称サブプログラムの仮引数が多相的である場合の考察が不十分かもしれない。宣言時の型での選択と実行時の型での選択の両方が必要になると思われる。

4. まとめ

本稿で提案する総称副プログラムに関わる言語拡張は次のとおりである。

- GENERIC 接頭辞と総称指定子: 副プログラムを総称副プログラムとして宣言する。
- 型宣言文の拡張: 総称副プログラムの仮引数の代替する型、種別、次元数を指定する。
- 引用仕様宣言の拡張: 総称副プログラムの明示的引用仕様を指定する。
- SELECT RANK 構文と SELECT TYPE 構文の拡張: 代替する型、種別、次元数に対応させた。

これまで、総称識別子のメカニズムが提供する総称名、演算子、代入、及び定義入出力は、ライブラリ利用者に大きな利便性をもたらしている。しかし、そのためにライブラリ提供者は、数十から数百の個別副プログラムを作成しなければならないことが多かった。そうでなければ、プロセッサに依存したプログラムを作成するか、実行時に決定コストや分岐コストを残したプログラムを作成するしかなかった。総称副プログラムは、個別副プログラムを記述するコードのサイズを大幅に削減するため、実行性能と移植性を損なうことなく、プログラミング・コストとメンテナンス・コストを削減することができる。

5. 謝辞

本稿を読んでプレゼンテーションの改善点を指摘してくれた John Reid に感謝する。SELECT 構文を拡張することで総称仮引数を扱うのは彼のアイディアによる。ユーザの立場から議論し、実践的な提案をしてくれた出川智啓氏及びユーザグループ Fortran-jp に感謝する。また、有益なコメントをくださった佐藤周行氏と林康晴氏、例や記述の改善点を指摘していただいた高田正之氏と鈴木敏弘氏に感謝する。Thomas Clune、Brad Richardson 及び Generics サブグループとの議論は、型指定子の改良に役立った。

修正履歴

Version 1.7→ 1.7J

- 全訳