Subject: Type parameters are distinguished along the wrong axis
From: Van Snyder

1 Introduction

Type parameters are required to be integers, and therefore can't fruitfully be used in non-integer initialization expressions. To allow this, type parameters need three attributes. NONKIND should be split between INITIALIZATION that's not used for KIND, and SPECIFICATION. If we freeze the NONKIND terminology now, it will be difficult to change it to initialization/kind/specification at a later date. This isn't a large change; most of it is in 4.5.1.1. The remainder consists mostly of replacing "nonkind" by "specification".

2 Edits

Edits refer to 99-007r2. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by immediately following text, while a page and line number followed by + indicates that immediately following text is to be inserted after the indicated line. Remarks for the editor are noted in the margin, or appear between [and] in the text.

[Editor: Add "integer scalar" before "expression".] 32:13

A type parameter is either a kind type parameter, an initialization type parameter, or a specification type parameter. 32:12

[Editor: Delete "nonkind type parameter" from the index and add "initialization type parameter" and "specification type parameter" to the index.]

[[Editor: Replace "an expression" by "a specification expression."] 32:28

[Editor: Replace "nonkind" by "specification" four times.] 32:28-33

The value of an initialization type parameter shall be specified by an initialization expression. An initialization type parameter may in turn be used within the derived type definition for the type. 32:34+

A typical use of an initialization type parameter is to specify the value of a primary in an initialization expression for a component.

NOTE $4.4\frac{1}{2}$

[Editor: Replace "nonkind" by "specification."] 32:41

[Editor: Replace "nonkind" by "specification."] 33:3

---

or *kind-type-param-name* 34:17+?

Constraint: The *kind-type-param-name* shall be the name of a kind type parameter of a derived type in which definition the *kind-param* appears. 34:22+?

[Editor: Replace "or nonkind" by ", initialization, or specification".] 40:40

---

type-param-attr-spec **is** KIND
or INITIALIZATION
or SPECIFICATION 42:24-25

[Editor: Delete "NONKIND attribute" from the index and add "INITIALIZATION attribute"

and "SPECIFICATION attribute" to the index.]

If a type parameter has the KIND attribute it is a **kind type parameter**. If it has the INITIALIZATION attribute it is an **initialization type parameter**. If it has the SPECIFICATION attribute it is a **specification type parameter** 45:13-46:33

A *type-param-attr-spec* may be used to specify explicitly whether a type parameter has the KIND attribute, the INITIALIZATION attribute or the SPECIFICATION attribute. If a type parameter is not given the KIND or INITIALIZATION attribute, either explicitly, or implicitly as described below, it has the SPECIFICATION attribute. It is allowed but not required to specify that a type parameter has the SPECIFICATION attribute.

Type parameters may themselves have type parameters, so long as there is no circular dependence between type parameters. If a type parameter has intrinsic type, and its parameters are not specified, the parameters of the type parameter have their default values.

NOTE 4.20$\frac{1}{2}$

The following is prohibited due to a circular dependence between two type parameters.

```
TYPE will_not_compile ( k, n )
  TYPE ( type_1 (k) ) :: n
  TYPE ( type_1 (n) ) :: k
  TYPE ( type_1 (k) ) :: comp = n
END TYPE will_not_compile
```

A kind type parameter shall have integer type. Any type parameter may be used as a primary in a specification expression (7.1.6) within the *derived-type-def*. A kind or initialization type parameter may be used in an initialization expression (7.1.7) within the *derived-type-def*.

With the exception stated below, a type parameter is implicitly given the KIND attribute [if it is used as a *kind-param* or] if it appears in the *derived-type-def* as a primary in an expression that is used to specify a kind parameter for a component, so long as that expression is not within an actual argument for a SELECTED_CHAR_KIND, SELECTED_INT_KIND, or SELECTED_REAL_KIND intrinsic function. With the same exception stated below, a type parameter that does not implicitly have the KIND attribute is implicitly given the INITIALIZATION attribute if it appears in the *derived-type-def* as a primary in an expression that is required to be an initialization expression. If a type parameter implicitly has the KIND or INITIALIZATION attribute, it shall not be specified to have a different type parameter attribute. *See tentative additions for page 34*

Note to J3

KIND will get more complicated if we allow PARAMETER within TYPE definitions.

NOTE 4.21

If IMPLICIT NONE is not in effect it may not be necessary to declare anything explicitly about a type parameter. Even if it is necessary to declare its type, it will frequently implicitly have the INITIALIZATION or KIND attributes. For example, consider

```
TYPE matrix ( k, n, v )
  REAL(k) :: element(n,n) = v
END TYPE matrix
```

If the implicit typing rules have not been changed, the parameters `k` and `n` are implicitly of type integer, and `v` is of type default real. The parameter `v` has the INITIALIZATION attribute because it is used in a context that requires it. The parameter `k` has the KIND attribute because it is used in a context that requires it.

NOTE 4.21 cont.

The following example uses explicit type parameter specifications.

```
TYPE humongous_matrix ( k, d, init )
  INTEGER, KIND :: k
  INTEGER(selected_int_kind(12)), SPECIFICATION :: d
  REAL(k), INITIALIZATION :: init
  REAL(k) :: element(d,d) = init
END TYPE humongous_matrix
```

In the following example, `dim` is explicitly declared to be a kind type parameter, even though it is not required by anything shown here. This would allow generic resolution of procedures distinguished only by values of `dim`.

```
TYPE general_point ( dim )
  INTEGER, KIND :: dim
  REAL :: coordinates ( dim )
END TYPE general_point
```

If a derived type that has a component that is a pointer to a (possibly different) derived type, the appearance of a type parameter of the containing type in an expression for a kind or initialization type parameter of the component implicitly declares it to be a kind or initialization type parameter, respectively, of the containing type only if the type definition for the type of the component preceeded that of the containing type.

NOTE 4.22

This rule is to avoid an indeterminacy caused by mutually recursive derived type definitions. For example

```
TYPE type_1 ( i )
  INTEGER, KIND :: i ! -- required because type_2 has not yet been
                     ! -- defined, and therefore whether it has a
                     ! -- kind type parameter is not yet known.
  TYPE ( type_2 ( i ) ), POINTER :: comp
END TYPE type_1

TYPE type_2 ( j )
  ! -- No explicit attribute specification needed for j.  J
  ! -- implicitly has the KIND attribute because it is used to
  ! -- specify a kind parameter of the previously defined type
  ! -- type_1.
  TYPE ( type_1 ( j ) ), POINTER :: comp
END TYPE type_2
```

This is not an issue except with pointer components, because a nonpointer component is required to be of a previously defined type.

[Editor: Replace “nonkind” by “specification” twice.]	59:3, 5
[Editor: Replace “nonkind” by “specification”.]	71:47
[Editor: Replace “nonkind” by “specification”.]	135:18
[Editor: Replace “non-kind” by “specification”.]	263:35