Subject: Parameterized module facility
From: Van Snyder

1 Number

TBD

2 Title

Parameterized module facility

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide a facility whereby a module or subprogram can be developed in a generic form, and then applied to any appropriate type.

6 Rationale

Many algorithms can be applied to more than one type. Many algorithms that can only be applied to one type can be applied to more than one kind. It is tedious, expensive, and error prone — especially during maintenance — to develop algorithms that are identical except for type declarations to operate on different types or kinds.

Generic modules are useful to package types together with their type-bound procedures, so that when they are instantiated, they are consistent. This cannot be guaranteed for parameterized types.

7 Estimated Impact

Moderate to extensive, depending on how it's done. The solution proposed here can be implemented mostly with changes in Section 11, and perhaps a few changes in Section 4. Estimated at J3 meeting 169 to be at 6 on the JKR scale.

8 Detailed Specification

Provide a variety of module called a **generic module**. A generic module is a template or pattern for generating specific instances. It has **generic parameters** but is otherwise structurally similar to a nongeneric module. A generic parameter can be a type, a data object, a procedure, a generic interface, a nongeneric module, or a generic module.

By substituting concrete values for its generic parameters, one can create an **instance of a generic module**. Entities from generic modules cannot be accessed by use association. Rather, entities can be accessed from instances of them. Instances of generic modules have all of the properties of nongeneric modules, except that they are always local entities of the scoping units in which they are instantiated.

Provide a means to create instances of generic modules by substituting concrete values for their generic parameters

Provide a means to access entities from instances of generic modules by use association.

It is proposed at this time that generic modules do not have submodules.

8.1 Priority for features

The features of generic modules depend primarily upon what varieties of entities are allowed as generic parameters.

The priority of what should be allowed for generic parameters and their corresponding instance parameters is, with most important first:

Generic parameter	Associated instance parameter
Type	Type
Data entity	Initialization expression Variable
Specific procedure	Specific procedure
Generic interface	Generic interface
Non-generic module	Non-generic module
Generic module	Generic module

To fit the proposal within the development schedule, it may be necessary to reduce the present scope of the proposal. If so, less-important features should be removed before more-important ones.

8.2 Definition of a generic module — general principles

A generic module may stand on its own as a global entity, or may be a local entity defined within a program, module or subprogram. It shall not be defined within another generic module. If it is defined within another scoping unit, instances of it access that scoping unit by host association. This is useful if a particular scoping unit is the only place where it's needed, or if instances need to share an entity such as a type, procedure or variable.

A second axis of simplification is to prohibit generic modules to be defined within other scoping units. If this is prohibited, instances should nonetheless not access scoping units where they are instantiated by host association, so as to preserve the possibility to extend to the functionality described here at a later time.

The MODULE statement that introduces a generic module differs from one that introduces a nongeneric module by having a list of generic parameter names.

The **interface** of a generic module is the list of the sets of characteristics of its generic parameters. The interface shall be explicitly declared, that is, the variety of entity of each generic parameter, and the characteristics required of its associated actual parameter when an instance is created, shall be declared. There shall be no optional parameters. Generic parameters and their associated instance parameters are described in detail in section 8.4 below.

Other than the appearance of generic parameters in the MODULE statement, and their declarations, generic modules are structurally similar to nongeneric modules, as defined by R1104:

R1104 *module* **is** *module-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-module-stmt

although it may be necessary to relax statement-ordering restrictions a little bit.

8.3 Instantiation of a generic module and use of the instance — general principles

An instance of a generic module is created by the appearance of a USE statement that refers to that generic module, and provides concrete values for each of the generic module's generic parameters. These concrete values are called **instance parameters**. The instance parameters in the USE statement correspond to the module's generic parameters either by position or by name, in the same way as for arguments in procedure references or component specifiers in structure constructors. The characteristics of each instance parameter shall be consistent with the corresponding generic parameter.

By substituting the concrete values of instance parameters for corresponding generic parameters, an

instance of a generic module is created, or **instantiated**. An instance of a generic module is a module, but it is a local entity of the scoping unit where it is instantiated. It does not, however, access by host association the scoping unit where it is instantiated. Rather, it accesses by host association the scoping unit where the generic module is defined.

Each local entity within an instance of a generic module is distinct from the corresponding entity in a different instance, even if both instances are instantiated with identical instance parameters.

A generic module shall not be an instance parameter of an instance of itself, either directly or indirectly.

A generic module may be instantiated and accessed in two ways.

- By instantiating it and giving it a name, and then accessing entities from the named instance by use association. Named instances are created by a USE statement of the form

  USE :: *named-instance-specification-list*

  where a *named-instance-specification* is of the form *instance-name* => *instance-specification*, and *instance-specification* is of the form *generic-module-name* (*instance-parameter-list*). In this case, the *only-list* and *rename-list* are not permitted — since this does not access the created instance by use association.

  Entities are then accessed from those instances by USE statements that look like R1109:

R1109	*use-stmt*	**is**	USE [[, *module-nature*] ::] *module-name* ■
			■ [, *rename-list*]
		or	USE [[, *module-nature*] ::] *module-name* , ■
			■ ONLY : [*only-list*]

  but with *module-name* replaced by *instance-name*.

- By instantiating it without giving it a name, and accessing entities from that instance within the same statement. In this case, the USE statement looks like R1109, but with *module-name* replaced by *instance-specification*.

In either case, a *module-nature* could either be prohibited, or required with a new value such as GENERIC or INSTANCE.

Alternatively, a new statement such as INSTANTIATE might be used instead of the above-described variations on the USE statement, at least in the named-instance case. In the anonymous-instance case it would be desirable to use the USE statement, to preserve functionality of *rename-list* and *only-list* without needing to describe them all over again for a new statement.

Since instances are essentially modules, but are always local entities within the program units where they are instantiated, it seems fatuous to prohibit nongeneric modules within other program units. It would be reasonable to limit the nesting depth, as we do for subprograms. For example, it would be reasonable to prohibit either a generic module or a nongeneric module to be defined within an internal or generic module.

8.4 Generic parameters and associated instance parameters

A generic parameter may be a type, a data entity, a specific procedure, a generic interface, a nongeneric module, or a generic module.

Declarations of generic parameters may depend upon other generic parameters, but there shall not be a circular dependence between them, except by way of pointer or allocatable components of generic parameters that are types.

8.4.1 Generic parameters as types

If a generic parameter is a type, it shall be declared by a type definition having the same syntax as a derived type definition. The type definition may include component definitions. The types and type parameters of the components may themselves be specified by other generic parameters. The type definition may include type-bound procedures. Characteristics of these type-bound procedures may depend upon generic parameters.

If the generic parameter is a type, the corresponding instance parameter shall be a type. If the generic parameter has components, the instance parameter shall at least have components with the same names, types, type parameters and ranks. If the generic parameter has type parameters, the instance parameter shall at least have type parameters with the same names and attributes. Type parameters of the instance parameter that correspond to type parameters of the generic parameter shall be specified by a colon, as though they were deferred in an object of the type — even if they are KIND parameters, and any others shall have values given by initialization expressions. If the generic parameter has type-bound specific procedures or type-bound generics, the corresponding instance parameter shall at least have type-bound specifics and generics that are consistent, except that if a specific procedure binding to the generic parameter has the ABSTRACT attribute the instance parameter need not have a specific binding of the same name because it is only used to provide an interface for a generic binding; it shall not be accessed by the specific name. Instance parameters that are intrinsic types shall be considered to be derived types with no accessible components. Intrinsic operations and intrinsic functions are available in every scoping unit, so it is not necessary to assume that intrinsic operations and intrinsic functions are bound to the type.

8.4.2 Generic parameters as data objects

If a generic parameter is a data object, it shall be declared by a type declaration statement. Its type and type parameters may be generic parameters. If it is necessary that the actual parameter to be provided when the generic module is instantiated shall be an initialization expression, the generic parameter shall have the KIND attribute, no matter what its type — even a type specified by another generic parameter.

If the generic parameter is a data object, the corresponding instance parameter's type, kind and rank shall be the same as specified for the generic parameter.

If the generic parameter is a data object with the KIND attribute, the corresponding instance parameter shall be an initialization expression.

If the generic parameter is a data object without the KIND attribute, the corresponding instance parameter shall be a variable. Every expression within the variable shall be an initialization expression. The instance has access to the variable by some newly-defined variety of association (or maybe by storage association) — instantiation does not create a new one with the same characteristics.

8.4.3 Generic parameters as procedures or generic interfaces

If a generic parameter is a procedure or a generic interface, its interface shall be declared explicitly. Its characteristics may depend upon generic parameters.

If the generic parameter is a procedure, the corresponding instance parameter shall be a procedure having characteristics consistent with the interface for the generic parameter, which interface may depend upon other generic parameters.

If the generic parameter is a generic interface, the corresponding instance parameter shall be a generic identifier, whose interface shall have at least specifics consistent with specific interfaces within the generic parameter's generic interface. The instance parameter need not have the same generic identifier as the generic parameter. If a specific interface within the generic parameter's generic interface has the ABSTRACT attribute, the instance parameter need not have a specific procedure with the same name, but it shall have a specific procedure with the same characteristics. In this case, the specific procedure within the generic parameter's generic interface cannot be accessed by the specified name as a specific procedure, either within an instance or from one by use association.

8.4.4 Generic parameters as generic or nongeneric modules

If a generic parameter is a generic module, The interface of that parameter shall be declared.

If the generic parameter is a generic module, the corresponding instance parameter shall be a generic module, having an interface consistent with the generic parameter.

If the generic parameter is a nongeneric module, the corresponding instance parameter shall be a nongeneric module, which may be an internal module or an instance of a generic module.

8.5 Instantiation of a generic module and use of the instance — fine points

If a generic module is defined within a module, it can have the PRIVATE attribute. This means it cannot be accessed by use association, which in turn means that it cannot be instantiated outside of the module where it is defined. Rather, it will be instantiated some fixed number of times within that module, which instances might or might not be accessible by use association. A similar situation holds, of course, if a generic module is defined within a scoping unit that is not a module.

If the generic module is an internal generic module, it shall be accessible in the scoping unit where the USE statement that instantiates it appears. This may require that it be made available by USE association from a module within which it is defined. That is, two USE statements may be necessary: One to access the generic module, and another to instantiate it.

If a generic module has a generic parameter that is a generic module, and the generic parameter is public, four USE statements might appear: One to access the generic module, one to instantiate it, one to access the generic parameter that is a generic module from that instance, and yet another to instantiate that generic module. This could be prohibited, for example by prohibiting generic parameters that are generic modules to be public, but why?

An instance parameter is accessible by use association from an instance of a generic module by using the identifier of the corresponding generic parameter, unless the generic parameter's identifier is private.

Where a module is instantiated, the *only* and *renaming* facilities of the USE statement can be used as well. Processors could exploit an *only-list* to avoid instantiating all of a module if only part of it is ultimately used. Suppose for example that one has a generic BLAS module from which one wants only a double-precision L2-norm routine. One might write `use BLAS(kind(0.0d0)), only: DNRM2 => GNRM2`, where `GNRM2` is the specific name of the L2-norm routine in the generic module, and `DNRM2` is the local name of the double-precision instance of it created by instantiating the module. If *only* is not used, every entity in the module is instantiated, and all public entities are accessed from the instance by use association, exactly as is currently done for a USE statement without an *only-list*.

If a named instance is created, access to it need not be in the same scoping unit as the instantiation; it is only necessary that the name of the instance be accessible. Indeed, the instance might be created in one module, its name accessed from that module by use association, and entities from it finally accessed by use association by way of that accessed name.

8.6 Examples of proposed syntax for definition

The following subsections illustrate how to define modules.

8.6.1 Sort module hoping for < routine

Here's an example of the beginning of a generic sort module in which the processor can't check that there's an accessible < operator with an appropriate interface until the generic module is instantiated. There's no requirement on the parameters of the generic type `MyType`. The only way the instance can get the < routine is if it is intrinsic, by host association from the scoping unit where the generic module is defined, or if it is bound to the type given by the instance parameter (recall that instances do not access by host association the scoping unit where they're instantiated). Aleks advocates that this one is illegal. The primary difference would be in the quality of message announced in the event `MyType` does not have a suitable < operator.

```
module Sorting ( MyType )
  type :: MyType
  end type MyType
  ....
```

8.6.2 Sort module with < specified by module parameter generic interface

The < operator is given by a generic parameter. When the module is instantiated, a generic identifier for an interface with a specific consistent with the `less` shown here, shall be provided as an instance parameter.

```
module SortingP ( MyType, Operator(<) )
  type :: MyType
  end type MyType
  interface operator (<)
    pure logical abstract function Less ( A, B ) ! "less" is purely an abstraction
      type(myType), intent(in) :: A, B
    end function Less
  end interface
  ....
```

The ABSTRACT attribute for the `less` function means that the associated instance parameter for `operator(<)` only needs to have a specific with the specified interface, but the name isn't required to be `less`. Indeed, `less` can't be accessed by that name within `SortingP` or by use association from an instance of `SortingP`.

The instance parameter corresponding to `operator(<)` need not have the same generic identifier. For example, if it's `operator(>)` (with the obvious semantics), the instantiated sort routine would sort into reverse order.

8.6.3 Sort module with < specified by type-bound generic interface

This illustrates a generic parameter that is a type that is required to have a particular type-bound generic. The type shall have a type-bound generic with a particular interface, but if entities are declared by reference to the name `MyType` or a local name for it after it is accessed from an instance, the specific type-bound procedure cannot be invoked by name; it can only be accessed by way of the type-bound generic. The `abstract` attribute does this. It's only allowed in the definitions of types that are generic parameters.

```
module SortingTBP ( MyType )
  type :: MyType
  contains
    procedure(less), abstract :: Less ! Can't do "foobar%less".  "Less" is only
      ! a handle for the interface for the "operator(<)" generic
    generic operator(<) => Less ! Type shall have this generic operator
  end type MyType
  ! Same explicit interface for "less" as in previous example
  ....
```

8.6.4 Module with type having at least a specified component

```
module LinkedLists ( MyType )
  type :: MyType
    type(myType), pointer :: Next! "next" component is required.
    ! Type is allowed to have other components, and TBPs.
  end type MyType
  ....
```

8.6.5 Module with type having separately-specified kind parameter

```
module LinkedLists ( MyType, ItsKind )
  type :: MyType(itsKind)
    integer, kind :: itsKind
  end type MyType
  integer, kind :: ItsKind
  ....
```

8.6.6 BLAS definition used in instantiation examples in 8.7

```
module BLAS ( KIND )
  integer, kind :: KIND
  interface NRM2; module procedure GNRM2; end interface NRM2
  ....
contains
  pure real(kind) function GNRM2 ( Vec )
  ....
```

8.6.7 Ordinary module with private instance count and internal generic module

```
module ModuleWithInternalGeneric
  integer, private :: HowManyInstances
  module InternalGeneric ( MyType )
    ! Instances of InternalGeneric access HowManyInstances by host association
    ....
```

8.7 Examples of proposed syntax for instantiation

The following subsections illustrate how to instantiate a generic module.

8.7.1 Instantiating a stand-alone generic module

Instantiate a generic module `BLAS` with `kind(0.0d0)` and access every public entity from the instance:

```
use BLAS(kind(0.0d0))
```

Instantiate a generic module `BLAS` with `kind(0.0d0)` and access only the `GNRM2` function from the instance:

```
use BLAS(kind(0.0d0)), only: GNRM2
```

Instantiate a generic module `BLAS` with `kind(0.0d0)` and access only the `GNRM2` function from the instance, with local name DNRM2:

```
use BLAS(kind(0.0d0)), only: DNRM2 => GNRM2
```

8.7.2 Instantiate within a module, and then use from that module

This is the way to get only one single-precision and only one double-precision instance of BLAS; instantiating them wherever they are needed results in multiple instances. This also illustrates two ways to make generic interfaces using specific procedures in generic modules. The first one creates the generic interface from specific procedures accessed from the instances:

```
module DBLAS
  use BLAS(kind(0.0d0))
end module DBLAS
module SBLAS
  use BLAS(kind(0.0e0))
end module SBLAS
module B
  use DBLAS, only: DNRM2 => GNRM2
  use SBLAS, only: SNRM2 => GNRM2
  interface NRM2
    module procedure DNRM2, SNRM2
  end interface
end module B
```

In the second one the generic module has the generic interface named NRM2 that includes the GNRM2 specific:

```
module DBLAS
  use BLAS(kind(0.0d0))
end module DBLAS
module SBLAS
  use BLAS(kind(0.0e0))
end module SBLAS
module B
  use DBLAS, only: NRM2    ! Generic; GNRM2 specific not accessed
  use SBLAS, only: NRM2, & ! Generic
    &    SNRM2 => GNRM2    ! Specific
end module B
```

8.7.3 Instantiate and access twice in one scoping unit, augmenting generic interface

```
module B
  use BLAS(kind(0.0d0)), only: NRM2    ! Generic; GNRM2 specific not accessed
  use BLAS(kind(0.0e0)), only: NRM2, & ! Generic NRM2 grows here
    &                SNRM2 => GNRM2    ! Specific
end module B
```

The method in 8.7.2 above might be desirable so as not accidentally to have multiple identical instances of BLAS in different scoping units.

8.7.4 Instantiate and give the instance a name, then access from it

```
! Instantiate BLAS with kind(0.0d0) and call the instance DBLAS, which is
! a local module.
use :: DBLAS => BLAS(kind(0.0d0))
! Access GNRM2 from the instance DBLAS and call it DNRM2 here
use DBLAS, only: DNRM2 => GNRM2
```

8.7.5 Instantiate two named instances in one module, then use one elsewhere

```
module BlasInstances
  ! Instantiate instances but do not access from them by use association
  use :: DBLAS => BLAS(kind(0.0d0)), SBLAS => BLAS(kind(0.0d0))
end module BlasInstances
module NeedsSBlasNRM2
  use BlasInstances, only: SBLAS  ! gets the SBLAS instance module, not its contents
  use SBLAS, only: SNRM2 => GNRM2 ! Accesses GNRM2 from SBLAS
end module  NeedsSBlasNRM2
```

8.7.6 Instantiate sort module with generic interface instance parameter

```
type :: OrderedType
  ...
end type OrderedType
interface operator (<)
  pure logical function Less ( A, B )
    type(orderedType), intent(in) :: A, B
  end function Less
end interface
! Notice relaxed statement ordering.
use SortingP(orderedType,operator(<)), only: OrderedTypeQuicksort => Quicksort
....
```

8.7.7 Instantiate sort module with TBP Less

```
  use SortingTBP(real(kind(0.0d0))), only: DoubleQuicksort => Quicksort
```

Notice that this depends on < being a "type-bound generic" that is bound to the intrinsic double precision type. Here's one with a user-defined type that has a user-defined type-bound < operator.

```
  type MyType
    ! My components here
  contains
    procedure :: MyLess => Less
    generic operator ( < ) => myLess
  end type MyType

  use SortingTBP(myType), only: MyTypeQuicksort => Quicksort
```

The interface for `less` is given in 8.6.2.

Notice that the USE statement comes *after* the type definition and the TBP's function definition.

8.8 Example of consistent type and TBP

This example illustrates how to create a type with type-and-kind consistent type-bound procedures, for any kind. This cannot be guaranteed by using parameterized types.

```
  module SparseMatrices ( Kind )
    integer, kind :: Kind
    type Matrix
      ! Stuff to find nonzero elements...
      real(kind) :: Element
    contains
      procedure :: FrobeniusNorm
      ....
    end type

  contains
    subroutine FrobeniusNorm ( TheMatrix, TheNorm )
      type(matrix), intent(in) :: TheMatrix
      real(kind), intent(out) :: TheNorm
      ....
    end subroutine FrobeniusNorm
    ....
  end module SparseMatrices

  ....

  use SparseMatrices(selected_real_kind(28,300)), & ! Quad precision
    & only: QuadMatrix_T => Matrix, QuadFrobenius => Frobenius, &
    &       QuadKind => Kind ! Access instance parameter by way of generic parameter

  ....

  type(quadMatrix_t) :: QuadMatrix
  real(quadKind) :: TheNorm

  ....

  call quadFrobenius ( quadMatix, theNorm )
```

9 History

03-264r1 m166
04-153 m167