Subject: Parameterized module facility for Fortran after 2003
From: Van Snyder

1 Problem

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.

2 Proposed solution

A generic programming facility for Fortran is proposed here. It is similar to an Ada generic package. Ada also provides for generic procedures, but if one has generic packages, generic procedures can be gotten by putting ordinary procedures inside of generic packages, so to keep this proposal simple, it does not include the equivalent of Ada generic procedures.

A generic package can be used to collect together types, procedures and other entities that need to be instantiated in a consistent way. It may be that one doesn't need all of the entities from an instance of it. It may be that one needs more than one instance of it in the same scope. In Ada, the **with** and **use** statements provide the necessary functionality to satisfy these desires.

The USE statement in Fortran and its relation to its module have all of the necessary functionality.

It is therefore proposed to implement a generic programming facility based on modules, by extending the syntax and semantics of MODULE and USE statements. No new statements or keywords are introduced. Extensions to the syntax of the interface block and type statement may be useful.

The MODULE statement is extended to allow an access specification (for internal modules) and to allow for **module parameters**. The USE statement is extended to allow for **instance parameters**.

A module that has parameters is a **parameterized module**. It is a template or pattern from which specific modules can be generated by substituting concrete entities for its parameters.

It is desirable to allow parameterized modules within other scoping units, first because that may be the only place they're needed, and second to allow instances to share entities. It would be unreasonable to require shared entities to be put into another module, made public, and accessed by use association. Once one has internal parameterized modules, it is possible but seems fatuous to prohibit internal nonparameterized modules, including parameterized modules. Therefore declaration constructs are extended to include internal modules, either parameterized or not. It would not be unreasonable to prohibit interal modules to contain internal modules, just as we prohibit internal subprograms from containing internal subprograms.

A module parameter can be an object of any type and kind, a type, a procedure, a generic identifier, or a module — including a parameterized module. The corresponding instance parameter can be an initialization expression, a type specification, a procedure name, a generic identifier, or a module, respectively. Each module parameter shall be declared. A statement of the form TYPE [[, *module-type-param-attr*] ::] *module-param-name-list* specifies that module parameters are types. The *module-type-param-attr* is of the form WITH (*binding-name-list*), which indicates that the specified generic or specific binding names are to be bound to the type. A statement of the form MODULE :: *module-param-name-list* specifies that a module parameter is a module.

Analogously to how dummy procedures may be declared, a procedure module parameter may be declared using a PROCEDURE statement or an interface body.

Each USE statement that has instance parameters creates a module that is a separate **instance** of a parameterized module, with instance parameters specified in the USE statement substituted for corresponding parameters of the specified module, and provides access by use association to entities of that instance. E.g. `use BLAS(dp), only: DAXPY => AXPY` creates an instance of `BLAS` with the instance parameter `dp` substituted for appearances of the corresponding module parameter, accesses an instance of `AXPY` by use association from that instance, and names it `DAXPY` in the instantiating scope. Instances are not created automatically to satisfy a need for an instance of an entity within a parameterized module.

Instance parameters correspond with module parameters by position or by keyword. There are no optional instance parameters. To allow for instance parameters that are intrinsic types, the syntax for type reference is extended to allow TYPE(*intrinsic-type*). An instance parameter that is a generic specifier may be associated with a procedure module parameter. To see that this flexibility is necessary, see the end of the example in 4.4.

The syntax of the USE statement is extended to give a name to an instance of a parameterized module, to allow accessing that instance by other USE statements. E.g. in `use :: DPBLAS => BLAS(dp); use DPBLAS, only: DAXPY => AXPY` the first USE statement creates the instance `DPBLAS` and the second accesses `DPBLAS` by use association. This is necessary if one has too many `only` names or too many renamings to fit onto a single statement.

Each entity within an instance of a parameterized module is separate from the corresponding entity in a different instance: Neither the specification parts nor the procedures are shared between different instances, and corresponding type definitions in different instances define different types, even if the instance parameters are identical. The SAVE attribute does *not* mean that a variable is shared between instances. If instances need to share an entity, they can access it by host or use association. If a recursive subprogram is created by an instance of a parameterized module, a variable within it that has the SAVE attribute is shared between recursively invoked instances of the procedure defined by that instance of that subprogram, not between procedures defined by different instances of that subprogram.

A parameterized module that is an internal module has access by host association to its containing scoping unit. An instance has access by host association to the containing scoping unit of its parameterized module's definition, but does not have access by host association to the containing scoping unit where it is instantiated.

An instance of a parameterized module shall not instantiate its parameterized module, either directly or indirectly. This includes prohibiting using a parameterized module as an instance parameter of itself if the corresponding module parameter is instantiated within the parameterized module. An internal module that is defined within a module shall not access its containing module by use association, either directly or indirectly.

The name of an internal module may be accessed by use association. This does not access the internal module by use association, or cause instantiation. An internal module may be instantiated or accessed by use association if its name is accessible. So to access an internal module from a module different from the one where it is defined, two USE statements are necessary; the first accesses the name of the internal module, and the second accesses the internal module or instantiates it. E.g., `use A, only: B; use B` or `use A, only: G; use G(myKind)`.

Alternatively, we could provide that an internal module may be directly instantiated or accessed by use association by qualifying its name with the name of its containing module(s), e.g., `use A%B` provides direct access by use association to the internal module `B` defined within the module `A`, while `use A%G(myKind)` instantiates `G` directly from `A` without separately accessing its name by use association.

No entity within a parameterized module is accessible by use association.

A USE statement that instantiates a parameterized module shall refer either to a global parameterized module or to an accessible internal parameterized module.

Neither an internal module nor a parameterized module shall have submodules.

If a procedure within a parameterized module has an operation, assignment, or input/output applied to objects of a type given by a module parameter, the operator, assignment or input/output shall be declared by using an interface block that is introduced by an *interface-stmt* of the form ABSTRACT INTERFACE *generic-spec*.

3 BNF for syntax extensions

The MODULE statement is extended:

R1105 *module-stmt* **is** MODULE *module-name* [(*module-param-name-list*)]

The USE statement is extended:

R1109 *use-stmt* **is** *module-reference*
or *module-instantiation*
R1109$\frac{1}{3}$ *module-reference* **is** USE [[, *module-nature*] ::] *module-name* [, *rename-list*]
or USE [[, *module-nature*] ::] *module-name* ,
■ ONLY : [*only-list*]
R1109$\frac{2}{3}$ *module-instantiation* **is** USE [:: [*instance-name* =>]] *module-name* ■
■ (*instance-param-list*) [, *rename-list*]
or USE [:: [*instance-name* =>]] *module-name* ■
■ (*instance-param-list*) , ONLY : [*only-list*]

The *module-name* in a USE statement might be the name of a global module, or an internal module that is defined in the same scope or accessed by host or use association.

R301 *instance-param* **is** [*module-param-name* =>] *instance-parameter*

R302 *instance-parameter* **is** *designator*
or TYPE (*derived-type-spec*)
or TYPE (*intrinsic-type-spec*)
or *procedure-name*
or *generic-spec*
or *module-name*

The *declaration-construct* is extended to provide for parameterized module instantiation, internal module definition, and declaration of module parameters:

R207 *declaration-construct* **is** *module-instantiation*
or *module*
or *derived-type-def*
or *module-type-param-decl*
or GENERIC [::] *module-param-name-list*
or MODULE :: *module-param-name-list*
or TYPE [, *module-type-param-attr* [::]] *module-param-name-list*

....

C201$\frac{1}{2}$ A *module-param-name* shall be the name of a parameter of the parameterized module in the scoping unit of which the *declaration-construct* appears.

R207$\frac{1}{2}$ *module-type-param-attr* **is** WITH (*tbp-name-list*)

The INTERFACE statement is extended to provide for abstract declaration of generic interfaces:

R1203 *interface-stmt* **is** [ABSTRACT] INTERFACE [*generic-spec*]

4 Aleksandar's example problems

4.1 A type with a type-bound procedure and the correct kind

```
  module Euclidean ( Kind )
    integer :: Kind

    type :: Euclidean_Point
      real(kind) :: Position(3)
    contains
      generic :: Translate => DoTranslate
    end type

```

```
contains

  subroutine DoTranslate ( Point, Translation )
    type ( Euclidean_Point ), intent(inout) :: Point
    type ( Euclidean_Point ), intent(in) :: Translation
    point%position = point%position + translation%position
  end subroutine DoTranslate

end module Euclidean

...
parameter :: R_SP = kind(0.0e0), R_DP = kind(0.0d0)
use euclidean(r_sp), only: Euclidean_Point_sp => euclidean_point
use euclidean(r_dp), only: Euclidean_Point_dp => euclidean_point
```

Notice that this *does not* use a parameterized type. Each time the `Euclidean_Point` type is instantiated from the `Euclidean` module, it's a new type, so it needs to be renamed if it's instantiated twice in the same scoping unit. Using a parameterized type would not guarantee that a type-bound `Translate` procedure with the appropriate kind of dummy arguments is available, but would give the opportunity to create objects for which it was *not* available.

We could use parameterized types here with an extension of the GENERIC statement in a type definition, wherein absence of the *binding-name-list* means "the *generic-spec* is the identifier of an interface body."

```
module :: Euclidean

  type :: Euclidean_Point(Kind)
    integer, kind :: Kind
    real(kind) :: Position(3)
  contains
    generic :: Translate
  end type

  module Translate_m ( Kind )
    integer :: Kind

    interface Translate
      module procedure DoTranslate
    end interface

    subroutine DoTranslate ( Point, Translation )
      type ( euclidean_point(kind) ), intent(inout) :: Point
      type ( euclidean_point(kind) ), intent(in) :: Translation
      point%position = point%position + translation%position
    end subroutine DoTranslate
  end module Translate_m

end module Euclidean

...
parameter :: R_SP = kind(0.0e0), R_DP = kind(0.0d0)
use euclidean, only: Euclidean_Point, Translate_m
use Translate_m(r_sp)
use Translate_m(r_dp)
```

Here you would need to be careful to instantiate enough instances of `Translate_m` to cover the kind parameters you're going to use to create `Euclidean_Point` objects. Each instantiation adds to the `Translate` generic interface body, so you would need to be careful not to instantiate `Translate_m` more than once with the same instance parameter. This could get icky if you have several kind parameters, which on some platforms are different and on others are the same. E.g., suppose you defined `R_SP = selected_real_kind(7)` and `R_DP = selected_real_kind(13)` on a 64-bit machine. Then you'd have two instances of DoTranslate, with identical characteristics, in the same generic.

If we allowed a USE statement inside of a type definition, one could do the following

```
module :: Euclidean

  private

  module, private :: Translate_m ( Kind )
    integer :: Kind
    subroutine DoTranslate ( Point, Translation )
      type ( euclidean_point(kind) ), intent(inout) :: Point
      type ( euclidean_point(kind) ), intent(in) :: Translation
      point%position = point%position + translation%position
    end subroutine DoTranslate
  end module Translate_m

  type, public :: Euclidean_Point(Kind)
    integer, kind :: Kind
    real(kind) :: Position(3)
  contains
    use translate_m(kind) ! Creates a "kind" instance of "DoTranslate"
    procedure :: DoTranslate
  end type

end module Euclidean

...
parameter :: R_SP = kind(0.0e0), R_DP = kind(0.0d0)
use euclidean, only: Euclidean_Point
type(euclidean_point(r_sp)) :: Point_SP
type(euclidean_point(r_dp)) :: Point_DP
```

This one is probably the most convenient one for users, but more work for a processor, because object declaration implies parameterized module instantiation. One might be tempted to wish in this case that the processor would cache instances of the module `Translate_m`. That's a step too far unless we provide something to instruct the processor that it's what the user wants: How does the processor know that the user doesn't *want* two instances (perhaps because it has a SAVE variable)?

4.2 `BLAS` for real and complex

```
module BLAS_1 ( Type )
  abstract interface operator ( + ) ! assumed to exist for intrinsic types
    pure type(type) function ADD_ ( X, Y )
      type(type), intent(in) :: X, Y ! Forward reference to declaration of "type"
    end function ADD_
  end interface
  type :: Type

contains
  function ADD ( X, Y ) result ( X_plus_Y )
```

```
      type(type), intent(in) :: X, Y
      type(type) :: X_plus_Y
      x_plus_y = x + y
    end function ADD
  end module BLAS_1
```

The interface for `operator ( + )` could be gotten by instantiating a module that consists only of such definitions, say by `use Numeric_Operators(type), only: operator(+)`. It indicates that the operator must be gotten from somewhere when the module is instantiated. Since instances don't access the environment of their instantiation by host association, the operator has to be bound to the type, with the specified interface. If a parameterized module has several parameters that are types, and declares a generic operation that involves several of them, the operation can be bound to any of the types, but not more than one (else the generic will be ambiguous).

```
  ...
  use BLAS_1(real(kind(0.0d0))), only: Add_DPR => Add
  use BLAS_1(complex(kind(0.0d0))), only: Add_DPC => Add
  interface ADD
    module procedure Add_DPR, Add_DPC
  end interface
```

In using this module it would be convenient to be able to put the USEs that instantiate `Add_DPR` and `Add_DPC` inside of the interface block:

```
  interface ADD
    use BLAS_1(real(kind(0.0d0))), only: Add_DPR => Add
    use BLAS_1(complex(kind(0.0d0))), only: Add_DPC => Add
  end interface
```

4.3 Generic stack

```
  module Stacks ( Type )
    type :: Type ! No requirements on this type

    type Stacks_t
      integer :: How_Many = 0
      type(type), allocatable, private :: Storage(:)
    contains
      procedure :: Initialize, Pop
    end type Stacks_t

  contains

    subroutine Initialize ( Stack, MaxSize )
      type(type), intent(out) :: Stack
      integer, intent(in) :: MaxSize
      allocate ( stack%storage(maxSize) )
    end subroutine Initialize
    ...
  end module Stacks

  type :: MyType
    ...
  end type MyType
```

```
use Stacks(myType), MyStack_t => stacks_t, Initialize_myType => initialize
type(myStack_t) :: MyStack
call myStack % initialize ( 100 )
```

4.4 Quicksort

```
module Quicksort_m ( Type )
  abstract interface operator ( < ) ! Assumed to exist for intrinsic types
    pure logical function Less ( X, Y )
      type(type), intent(in) :: X, Y ! Forward reference to declaration of "type"
    end function Less
  end interface
  type :: Type

contains

  subroutine Quicksort ( A )
    type ( type ), inout :: A(:)
    ...
    if ( A(i) < A(j) ) then
      ...
  end subroutine Quicksort
end module Quicksort_m

...
use Quicksort_m(type(myType)), only: Quicksort
```

The interface for `operator ( < )` could be gotten by instantiating a module that consists only of such definitions, say by `use Ordered_Operators(type), only: operator(<)`.

Alternatively, if < isn't expected to be bound to the `Type` module parameter

```
module Quicksort_m ( Type, Less )
  type :: Type
  interface operator ( < )
    pure logical function Less ( X, Y )
      type(type), intent(in) :: X, Y
    end function Less
  end interface

contains

  subroutine Quicksort ( A )
    type ( type ), inout :: A(:)
    ...
    if ( A(i) < A(j) ) then
      ...
  end subroutine Quicksort
end module Quicksort_m

...
use Quicksort_m(type(integer),operator(<)), only: Quicksort_int
use Quicksort_m(type(myType), myLessFunc), only: Quicksort_myType
```