Subject: Second draft of edits for parameterized modules
From: Van Snyder
Reference: 03-264r1, 04-153, 04-383r1, 05-107, 05-181, WG5/N1626-J3-014

1 Introduction

Parameterized modules were put onto the WG5 "allowed" list at Delft, where the project was descoped from the proposal in 04-383r1, restricting parameterized modules to be global entities, and module parameters to be only types and data entities. Instance parameters that correspond to module parameters that are data entities are restricted to be initialization expressions.

All that needs to be said about submodules is that parameterized modules do not have them.

The editor's guidance will be needed concerning the preferred method to specify where the edits apply.

2 Detailed specifications as revised at Delft

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.

The varieties of entities allowed as generic parameters are:

Generic parameter	Associated instance parameter
Type	Type
Data entity	Initialization expression

2.1 Definition of a generic module — general principles

A generic module shall stand on its own as a global entity. Instances do not access scoping units where they are instantiated by host association. 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 2.3 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.

2.2 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.

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 *use-stmt*, 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.

2.3 Generic parameters and associated instance parameters

A generic parameter may be a type or a data entity.

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.

2.3.1 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 within the generic module 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.

2.3.2 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. It is necessary that the actual parameter to be provided when the generic module is instantiated shall be an initialization expression, so the generic parameter shall have the KIND attribute, no matter what its type - even a type specified by another generic parameter.

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

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.

3 Questions

(1) This paper specifies that for purposes of correspondence between instance parameters and module parameters, intrinsic operations are considered to be generics that are type-bound to the intrinsic types. Thus if a type module parameter requires its corresponding instance parameter to have a type-bound generic < operator, an intrinsic type is sufficient. Should this be done differently? How?

(2) This paper does not specify that intrinsic procedures are considered to be bound to types. If a type module parameter requires that its corresponding instance parameter has, for

example, a type-bound ABS function, would REAL be a suitable instance parameter? What about something like SPREAD, that would have to be considered to be bound to every type?

(3) Some parameterized modules might work for some spectrum of types, but not be expected to work for others — say for REAL and INTEGER. Would it be useful to have syntax to specify that an instance parameter that corresponds to a type module parameter shall be one of a specified list of types?

4 Edits

Edits refer to 04-007. 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 associated text, while a page and line number followed by + (-) indicates that associated text is to be inserted after (before) the indicated line. Remarks are noted in the margin, or appear between [and] in the text.

[Editor: Add an item to the new-features list in the Introduction:] xiii

(1) Module enhancements: parameterized modules (allows a module to be developed independently of a specific type, and then instantiated with any type that satisfies requirements established by the parameterized module).

[Editor: Replace the first production for *specification-part* (R204):] 9:38

R204 *specification-part* **is** *global-use-association-stmt*

[Editor: Add a right-hand side at the end of those for *implicit-part-stmt* (R206):] 10:6+

or *other-use-stmt*

[Editor: Add right-hand sides for *declaration-construct* (R207), in alphabetical order:] 10:11+

or *module-param-decl*

or *other-use-stmt*

[Editor: Replace Table 2.1. Notice that the erstwhile row 4 is gone — because it was wrong!] 14

Table 2.1: **Requirements on statement ordering**

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE,
MODULE, or BLOCK DATA statement</td></tr>
<tr><td colspan="3">Global use association statements</td></tr>
<tr><td colspan="3">IMPORT statements</td></tr>
<tr><td rowspan="3">FORMAT
and
ENTRY
statements</td><td rowspan="2">Other USE statements and
PARAMETER statements</td><td>IMPLICIT statements</td></tr>
<tr><td>Derived-type definitions,
interface blocks,
type declaration statements,
DATA statements,
enumeration definitions,
procedure declarations,
specification statements,
and statement function statements</td></tr>
<tr><td colspan="2">Executable constructs and DATA statements</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal subprograms or module subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

We don't add parameterized modules to Table 2.2 because they don't fit. Should they be added? Keep in mind we need to try to shoehorn submodules, too. I'd be happy to delete Table 2.2. 14 *J3 question*

[Editor: Insert a constraint immediately before *private-or-sequence* (R432):] 45:21+

C429a (R430) If *type-name* is a *module-param*, *derived-type-def* shall be a *module-param-decl*.

[Editor: Insert a third constraint after *sequence-stmt* (R434):] 46:15+

C433a (R429) A *sequence-stmt* shall not appear if *type-name* is a *module-param*.

[Editor: Insert a constraint immediately before *attr-spec* (R503):] 71:17+

C503a (R502) A *module-param* shall not be declared with the CLASS keyword.

[Editor: Insert a new right-hand side for *attr-spec* (R503) between EXTERNAL and INTENT:] 71:22+

or INITIALIZATION

[Editor: Insert a new constraint immediately before *object-name* (R505):] 72:12+

C504a (R504) If *object-name* is *module-param*, *type-declaration-stmt* shall be a *module-param-decl*.

[Editor: Within the first constraint after *object-name* (R505) (C505), after "data object" insert "or a data entity module parameter. If it is a data entity module parameter it shall be a scalar, explicit-shape array, or a deferred-shape array".] 72:14

[Editor: Within the ninth constraint after *null-init* (R507) — the one that begins "The PARAMETER attribute . . . " — insert ", a module parameter" after "function".] 72:33

[Editor: Immediately after the ninth constraint after *null-init* (R507) — the one that begins "The PARAMETER attribute . . . " — insert two constraints:] 72:33+

C514a (R501) The INITIALIZATION attribute shall be specified if and only if *object-name* is a *module-param*.

C514b (R501) If the INITIALIZATION attribute is specified, the ALLOCATABLE, ASYNCHRONOUS, EXTERNAL, INTRINSIC, POINTER or VOLATILE attribute shall not be specified.

[Editor: Within the twelfth constraint after *null-init* (R507) — the one that begins "The SAVE attribute . . . " — insert ", a module parameter" after "result".] 72:39

[Editor: Within the nineteenth constraint after *null-init* (R507) — the one that begins "*initialization* shall not appear . . . " — insert ", a module parameter" after "result".] 73:11

[Editor: At the end of the zillionth constraint after *null-init* (R507) — the one that begins "If a *language-binding-spec* with a NAME= . . . " — insert "that does not declare a module parameter" after "*entity-decl*".] 73:31

[Editor: At the end of the zillionth plus two constraint after *null-init* (R507) — the second one that begins "The PROTECTED attribute . . . " — insert "and is not a module parameter" after "block".] 73:34

[Editor: Replace the first paragraph of **5.1.2.5.3 Deferred-shape array**:] 79:20

A **deferred-shape array** is an allocatable array, an array pointer, or a data entity module parameter.

[Editor: Immediately before *deferred-shape-spec* (R515), insert a new paragraph:] 79:26+ New ¶

The bounds, and hence shape, of a data entity module parameter in an instance (11.2.2) of a parameterized module are determined from the instance parameter associated with the module parameter where the parameterized module is instantiated.

[Editor: Before "A **module**" in the first sentence of the first paragraph of 11.2 insert the following sentences within the same paragraph:] 250:3

A module is characterized by two independent factors. One is whether it has parameters; the other is whether it is provided as an inherent part of the processor. A module that has parameters is a parameterized module (11.2.1). The term *module*, where not qualified by the adjective *parameterized*, indicates a module that does not have parameters.

[Editor: After "unit." at the end of the first sentence of the first paragraph of 11.2 insert the following sentence within the same paragraph:]

A **parameterized module** (11.2.1) is a pattern or template that can be used to create an instance (11.2.2) that is a module.

[Editor: Replace *module-stmt* (R1105):] 250:11

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

[Editor: Between the last constraint and the first note in **11.2 Modules**, insert a new paragraph:] 250:25+ New ¶

If a *module-param-list* appears in a *module-stmt*, the module it introduces is a parameterized module (11.2.1).

[Editor: Insert the following in **11.2 Modules** between Note 11.6 and the paragraph that begins "If a procedure declared. . . ":] 250:26-

R1108a *module-param* **is** *name*

R1108b *module-param-decl* **is** *type-declaration-stmt*
or *derived-type-def*

C1107a (R1105) Every *module-param* shall be declared by a *module-param-decl*.

C1107b (R1108b) A *module-param-decl* shall not appear except in the specification part of a module.

[Editor: Insert new subclauses before **11.2.1 The USE statement and use association** and renumber subsequent ones (TEX-o-matic):] 251:4+

11.2.1 Parameterized modules

A parameterized module is a module that has a *module-param-list* in its *module-stmt*. It serves as a template or pattern for creating instances (11.2.2) by substituting entities for its parameters. Parameters may be data entities or types.

The **interface** of a parameterized module determines how it can be instantiated. It consists of the names of its parameters and their characteristics as module parameters.

The characteristics of a data entity module parameter are its type, type parameters, shape, the exact dependence of its type, type parameters or array bounds on other entities, whether the shape is assumed, and which if any of its type parameters are assumed.

The characteristics of a type module parameter are its type parameters, its component names, the characteristics or its components, the interfaces of its type-bound procedures, the generic identifiers of its generic bindings, and which type-bound procedures are bound to each generic binding.

Every data entity module parameter shall be declared by a *type-declaration-stmt*. Every type module parameter shall be declared by a *derived-type-def*.

11.2.2 Instances of parameterized modules

An **instance** of a parameterized module is a nonparameterized module that is created by a USE statement that specifies entities to be substituted for the module parameters of the parameterized module. It is a local entity of the scoping unit in which it is instantiated, but it does not access that scoping unit by host association. An entity other than a module parameter in one instance is distinct from the corresponding entity in a different instance. A module parameter in one instance is distinct from the corresponding module parameter in a different instance if and only if the instance parameters corresponding to those module parameters are distinct.

[Editor: Replace the subclause heading and the first paragraph of **11.2.1 The USE statement and use association**:] 251:5-8

11.2.4 The USE statement

The **USE statement** specifies use association or creates an instance of a parameterized module. A USE statement is a **module reference** to the module it specifies. A module shall not reference itself, either directly or indirectly.

[Changing the subclause heading may entail either creating a label `D11:The USE statement and use association` or finding and changing references to that label to refer to the revised section heading.]

[Editor: Replace *use-stmt* (R1109):] 251:18-20

R1109 *global-use-association-stmt* **is** USE [[, *module-nature*] ::] *module-name* ■
■ *module-ref-specialization*
R1109a *other-use-stmt* **is** USE [[, *module-nature*] ::] *module-name* ■
■ [(*instance-parameter-spec-list*)] *module-ref-specialization*
or USE [[, *module-nature*] ::] *instance-name* => ■
■ *module-name* (*instance-parameter-spec-list*)

[Editor: Between *module-nature* (R1110) and *rename* (R1111) insert new syntax rules:] 251:22+

R1110a *module-ref-specialization* **is** [, *rename-list*]
or , ONLY : [*only-list*]
R1110b *instance-parameter-spec* **is** [*keyword* =] *instance-parameter*
R1110c *instance-parameter* **is** *initialization-expr*
or *declaration-type-spec*

[Editor: Before the third constraint after *only-use-name* (R1113) — the one that begins "A scoping unit . . . " — insert new constraints:] 251:32+

C1109a (R1109) The *module-name* shall be the name of a nonparameterized module.

C1109b (R1109a) The *module-name* shall be the name of a parameterized module or the *instance-name* of an instance of a parameterized module that is accessed by host association, previously accessed within the same scoping unit by use association, or previously instantiated within the same scoping unit.

[Editor: Before the fourth constraint after *only-use-name* (R1113) — the one that begins "OPERATOR . . . " — insert new constraints:] 251:34+

C1110b (R1109a) An *instance-parameter-spec-list* shall appear if and only if *module-name* specifies a parameterized module.

C1110c (R1110b) The *keyword* = shall not be omitted from an *instance-parameter-spec* unless it is omitted from each preceding *instance-parameter-spec* in the *instance-parameter-spec-list*.

C1110d (R1110b) Each *keyword* shall be the name of a parameter of the module specified by *module-name*.

[Between the constraints and ordinary normative text in **11.2.1 The USE statement and use association** — before the paragraph that begins "A *use-stmt* without . . . " — insert a new subclause:] 252:7+

11.2.4.1 Instantiation of parameterized modules

A USE statement in which an *instance-parameter-spec-list* appears creates an **instance** of a parameterized module by substituting entities for corresponding module parameters. The *instance-parameter-spec-list* identifies the correspondence between the instance parameters specified and the parameters of the module. This correspondence may be established either by keyword or by position. If an instance parameter keyword appears, the instance parameter corresponds to the module parameter whose name is the same as the instance parameter keyword. In the absence of an instance parameter keyword, the instance parameter corresponds to the module parameter occupying the corresponding position in the module parameter list; that is, the first instance parameter corresponds to the first module parameter, the second instance parameter corresponds to the second module parameter, etc.

C1115a (R1109a) Every instance parameter specified in a USE statement shall correspond with a module parameter of the specified module, and every module parameter of the specified module shall have a corresponding instance parameter.

C1115c (R1109a) An instance parameter that corresponds to a data entity module parameter shall be an initialization expression that has the same characteristics as the characteristics of its corresponding module parameter.

C1115d (R1109a) An instance parameter that corresponds to a type module parameter shall be a type that at least has components that have the same names and characteristics as the public components of the type module parameter, and shall at least have type-bound procedures and generic

bindings that have the same identifiers and characteristics as the public type-bound procedures and generic bindings of the type module parameter.

NOTE 11.8$\frac{1}{3}$

Intrinsic types do not have components.

An instance parameter that corresponds to a type module parameter may have additional components or type-bound procedures or generic bindings. For purposes of correspondence between instance parameters and module parameters, intrinsic operations are considered to be type-bound procedures of intrinsic types.

NOTE 11.8$\frac{2}{3}$

It is possible for a type module parameter to require its corresponding instance parameter to have a generic binding with particular interfaces without requiring its type-bound procedures to have specified names by making the generic binding of the type module parameter public and the type-bound procedures of the generic binding private.

If the USE statement has an *instance-name* it creates an instance named by the *instance-name* but does not access it by use association. The created instance a module that may be accessed by use association. If the USE statement does not have an *instance-name* it creates an instance that does not have a name, and accesses it by use association. Since the instance does not have a name, it cannot be referenced by a different USE statement.

[Editor: Then insert a subclause title for the existing normative text:]

11.2.4.2 Use association

[Editor: Within the second paragraph of **11.2.1 The USE statement and use association** — the one that begins "The **USE statement** provides . . . " – replace "The **USE statement**" at 251:9 by "**Use association**". Then move that second paragraph (at 251:9-17) and the subsequent Note (11.7) to here. Then insert the following new paragraph:]

A USE statement without an *instance-parameter-spec-list* specifies use association.

[Editor: before *proc-language-binding-spec* (R1225) insert a new constraint:] 279:25+

C1235a (R1224) The *function-name* shall not be the name of a function that has the ABSTRACT prefix.

[Editor: Add a new right-hand side for *prefix-spec* (R1228) (and perhaps alphabetize the ones already there):] 280:3+

or ABSTRACT

[Editor: Before *suffix* (R1229) insert a new constraint:] 280:7+

C1242a (R1227 A *prefix* shall not specify ABSTRACT unless it is within a *function-stmt* or *subroutine-stmt* that introduces an interface body within an interface block that declares the interface of a procedure bound to a type that is a module parameter (11.2.1).

[Editor: Before *dummy-arg* (R1233) insert a new constraint:] 282:10+

C1247a (R1232) The *subroutine-name* shall not be the name of a subroutine that has the ABSTRACT prefix.

[Editor: Within the first item in the first numbered list in **16.2 Scope of local identifiers**, insert "module parameters," before "dummy".] 406:5

[Editor: If we keep the glossary, insert the following glossary items in alphabetical order:] 430:35+

instance of a parameterized module (11.2.2, 11.2.4.1) A module that is created by substituting entities for a parameterized module's module parameters.

---

interface of a parameterized module (11.2.1) : The names of the modules parameters and their 431:6+

characteristics as module parameters.

---

parameterized module (11.2.1) : A module whose initial statement has a *module-param-list*. It serves as a template for creating instances by substituting entities for its parameters. 433:3+

[Editor: Insert the following subclauses before **C.9 Section 12 notes**:] 477:29+

C.8.4 Parameterized modules (11.2.1)

A parameterized module is a template that may be used to create specific instances by substituting entities for its module parameters.

C.8.4.1 Examples of definition of parameterized modules

C.8.4.1.1 Sort module with < intrinsic or type bound

This is an example of the beginning of a generic sort module in which the < operator with an appropriate interface is intrinsic or is bound to the type of its operands. In general, the processor cannot check that one with an appropriate interface is accessible until the module is instantiated. There is no requirement on the parameters of the type module parameter `MyType`. The quality of message announced in the event `MyType` does not have a suitable < operator is less than would be the case if the < operator were required to be bound to the type of a type module parameter.

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

C.8.4.1.2 Sort module with < specified by type-bound generic interface

This illustrates a module parameter that is a type that is required to have a particular type-bound generic identifier. The type shall have a type-bound generic identifier 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 `private` attribute does this.

```
module SortingTBP ( MyType )
  type :: MyType
  contains
    procedure(less), private :: 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
  abstract interface
    logical function Less ( A, B )
      type(myType), intent(in) :: A, B
    end function Less
  end interface
  ....
```

C.8.4.1.3 Module with type module parameter 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
  ....
```

C.8.4.1.4 Module with type module parameter having separately-specified kind parameter

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

C.8.4.1.5 BLAS definition used in instantiation examples in C.8.4.2

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

C.8.4.2 Examples of instantiation of parameterized modules

The following subclauses illustrate how to instantiate a parameterized module.

C.8.4.2.1 Instantiating a parameterized module

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

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

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

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

Instantiate a parameterized 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
```

C.8.4.2.2 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 parameterized 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 parameterized 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
```

C.8.4.2.3 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 C.8.4.2.2 above might be desirable so as not accidentally to have multiple identical instances of BLAS in different scoping units.

C.8.4.2.4 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
```

C.8.4.2.5 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
```

C.8.4.2.6 Instantiate sort module with type-bound Less procedure

```
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, private :: MyLess => Less
    generic operator ( < ) => myLess
  end type MyType

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

The interface for `less` is given in C.8.4.1.2. The name of the specific type-bound procedure bound to < need not be `less`.

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

C.8.4.2.7 Example of consistent type and type-bound procedure

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, initialization :: 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 )
```