Date:	31 October 1998
To:	J3
From:	Van Snyder
Subject:	Integrating abstract procedure interfaces, procedure pointers, dummy procedures and type-bound procedures.
References:	98-104

1 Background

Several problems have been noticed that would be reduced or eliminated if modifications along the lines proposed in this paper were adopted:

- In order to get a useful (non-sequence) derived type definition into an interface body, one must `USE` the module in which it appears. It is not possible to `USE` a module from within itself, but in order for a procedure to have access to private components, it must be within the same module as the type. If a procedure that needs access to private components of a type has a dummy procedure argument that has an argument of the type, it is not possible to write an explicit interface for the dummy procedure.
- Some people have expressed a preference to write the bodies of type-bound procedures inside of type declarations. Other people refuse to be required to do so.
- The use of the `NULL` intrinsic function to define abstract type-bound procedures is not parallel to the use of any other intrinsic function.

2 Proposal

- Define the concepts of "abstract procedure" – a procedure that explicitly has no body, and cannot be invoked – and "incomplete procedure" – a procedure for which the characteristics and body are specified at different places (see also 98-104 for other applications of the latter).

  In what follows, a "concrete procedure" means either a complete or incomplete procedure, but not an abstract one.
- Refer to abstract or concrete procedures in procedure declaration statements (section 5.2 in 98-007r3). Allow referring to concrete procedures because in many applications, one will have access to a collection of procedures, e.g. from a module, that might be targets of a pointer. It would be easier to allow one of them to serve as the "template" for the pointer rather than to require writing an additional abstract interface, identical to the interfaces in the set of concrete procedures that are expected to be targets of the procedure pointer.

  Additional features not present in 98-007r3: Allow a list of concrete procedures in the position in a procedure declaration statement (see 4.5.1.5) where the interface is expected. These procedures shall have the same characteristics (or maybe not?).

  - In a procedure pointer declaration it means that only one of those procedures can be assigned to the pointer. If the *target* in a pointer assignment is a pointer, is is required to have been declared with a list of concrete procedures, naming a (not necessarily proper) subset of the set of procedures allowed for the *pointer*.

   - In a dummy procedure declaration in means that only one of those procedures can be an actual argument associated to the dummy argument. During association of an actual argument pointer to a dummy procedure, the actual argument pointer is required to have been declared with a list of concrete procedures, naming a (not necessarily proper) subset of the set of procedures allowed for the dummy procedure.

  This is well-defined, but perhaps a bit unexpected: A `PROCEDURE` declaration statement with no interface allows any procedure to be pointer or argument associated; with one interface, abstract or concrete, it allows any procedure with the same interface; with more than one interface, they must all be concrete, and only those listed are allowed.

  There are several alternatives to support the desire to have several procedure pointers or dummy procedures that are constrained to the same set of concrete procedures:

  1. The user could repeat the list in each procedure pointer or dummy procedure declaration.
  2. The user could specify a derived type that consists of a procedure pointer component with the desired interface, and use it instead of repeating the list in numerous procedure pointer declarations. This subterfuge is clumsy, however, in the case of dummy procedures (a procedure pointer assignment is needed for each dummy procedure argument, possibly before each call).
  3. The standard could allow to encapsulate the list of procedures in an interface specification, to which the user could refer from procedure pointer or dummy procedure declarations.

     The interface specification *could* look like a generic interface block, but it would change the semantics of generic interfaces: it would be necessary to allow the characteristics of specific procedures in the interface to intersect. It would thereby not be possible to check at the point an interface block is declared whether invocations of the generic thus defined would inevitably be unambiguous; the checks would necessarily be applied to the invocations.

     It would seem to be better to use a different syntax of interface specification, but this is just chipping around the edges of the real problem, which is the lack of a comprehensive type algebra that included "procedure declaration" as one of the kinds of user-definable types.
  4. The standard could define a comprehensive type algebra. This is far beyond the scope of work allocated for the current effort.

  My preferences, in order, are 1 and 2, 4, don't do this feature, 3.

- Delete the interface-block-based definition of abstract interfaces (section 12.3.2.1.4 in 98-007r3).

- If we ever do generic type-bound procedures, allow reference to abstract procedures from within interface blocks, for the purpose of defining generic abstract interfaces for abstract types.

- Change the syntax of a `PROCEDURE` statement used to define a type-bound procedure to be the same as a procedure declaration statement (see section 4.5.1.5). Allow either a concrete or abstract procedure to specify the interface. The distinction between a pointer declaration and a procedure binding would depend on the presence or absence of the `POINTER` attribute, respectively.

- Allow definition of abstract procedures, incomplete procedures (interface information only) or complete procedures within or without type definitions.
- Allow definition of abstract procedures and incomplete procedures (interface information only) in modules before `CONTAINS`, with the bodies for incomplete procedures after `CONTAINS`.

3 Syntax ideas

To define an abstract or incomplete procedure, use the same sort of information as would be put into an interface body, with the crucial difference that abstract and incomplete procedure definitions *do* access the host environment by host association.

Alternatives:

1. Begin with an ordinary procedure header, and finish the procedure declaration by writing `ABSTRACT` or `SEPARATE`. Can't be put before `CONTAINS` because of the ambiguity of `REAL FUNCTION F ( N )` in fixed form – unless we allow :: notation in procedure headers.
2. Put `ABSTRACT` or `SEPARATE` in the procedure header.
3. Use `ABSTRACT` or `SEPARATE` to introduce "divisions" of the module or type definition, similar to the way `CONTAINS` is presently used.
4. Use `ABSTRACT ... END ABSTRACT` and `SEPARATE ... END SEPARATE`.

It is desirable to indicate, when the procedure body is written, that it is the completion of an earlier incomplete procedure. (Compilers can figure this out, but it's tedious and error prone for humans to do so.) I suggest adding a keyword to the procedure header, e.g. `COMPLETING` or `CONTINUING`. Need to decide whether it's allowed, prohibited or required to repeat argument and result characteristics where the procedure is completed.

4 Examples

4.1 Procedure pointers

Abstract procedure definitions from Note 12.10, re-written in two of the forms advocated here:

```
FUNCTION :: REAL_FUNC ( X ) or   ABSTRACT FUNCTION REAL_FUNC ( X )
  REAL, INTENT(IN) :: X            REAL, INTENT(IN) :: X
  REAL :: REAL_FUNC                REAL :: REAL_FUNC
ABSTRACT                         END FUNCTION REAL_FUNC

SUBROUTINE :: SUB ( X )     or   ABSTRACT SUBROUTINE SUB ( X )
  REAL, INTENT(IN) :: X            REAL, INTENT(IN) :: X
ABSTRACT                         END SUBROUTINE SUB
```

The examples in note 5.21 wouldn't change.

4.2 Dummy procedure argument

If a procedure needs access to private components of a type, and it has a dummy procedure that has an argument of that type, then it is impossible to write an explicit interface for the dummy procedure in Fortran 95. It's easy here:

```
MODULE M
  TYPE :: T
    PRIVATE
    ! ...
  END TYPE T

  SUBROUTINE :: IDUM ( X )     or   ABSTRACT SUBROUTINE IDUM ( X )
    TYPE(T), INTENT(IN) :: X          TYPE(T), INTENT(IN) :: X
  ABSTRACT                          END SUBROUTINE IDUM

CONTAINS

  SUBROUTINE SUB ( A, DUM )
    TYPE(T), INTENT(...) :: A
    PROCEDURE(IDUM) :: DUM
    ...
  END SUBROUTINE SUB
  ...
END MODULE M
```

4.3 Abstract type (type with abstract type-bound procedure)

4.3.1 Abstract procedure declared outside of the type

```
SUBROUTINE :: A_SUB ( X )   or   ABSTRACT SUBROUTINE A_SUB ( X )
  REAL, INTENT(IN) :: X            REAL, INTENT(IN) :: X
ABSTRACT                         END SUBROUTINE A_SUB

TYPE :: T ! I think TYPE, ABSTRACT :: T would be a good idea
          ! Otherwise, the only way one knows it's abstract is to
          ! look at the definitions of the interfaces of all of the
          ! procedure bindings.
  ... ! Data components
CONTAINS
  PROCEDURE(A_SUB) :: T_SUB
! Compare to present PROCEDURE :: T_SUB => NULL(A_SUB)
END TYPE T
```

4.3.2 Abstract procedure declared inside of the type

```
TYPE :: T
  ... ! Data components
CONTAINS
  SUBROUTINE T_SUB ( X )      or   ABSTRACT SUBROUTINE T_SUB ( X )
    REAL, INTENT(IN) :: X            REAL, INTENT(IN) :: X
  ABSTRACT                         END SUBROUTINE SUB
END TYPE T
```

4.4 Concrete type (type with concrete type-bound procedure)

4.4.1 Concrete procedure declared outside of the type

```
MODULE M
  TYPE :: T
    ... ! Data components
  CONTAINS
    PROCEDURE(C_SUB) :: T_SUB
  ! Compare to present PROCEDURE :: T_SUB => C_SUB
  END TYPE T

CONTAINS

  SUBROUTINE C_SUB ( X )
    REAL, INTENT(IN) :: X
    ...! Body of procedure
  END SUBROUTINE C_SUB
END MODULE M
```

4.4.2 Concrete procedure defined inside of the type

```
TYPE :: T
  ... ! Data components
CONTAINS
  SUBROUTINE T_SUB ( X )
    REAL, INTENT(IN) :: X
    ...! Body of procedure
  END SUBROUTINE A_SUB
END TYPE T
```

4.4.3 Concrete type containing incomplete procedure declaration

The purposes proposed here are identical to those proposed in 98-104: Define a procedure interface at one point, and the body elsewhere.

```
MODULE M
  TYPE :: T
    ... ! Data components
  CONTAINS
    SUBROUTINE SUB ( X )        or   SEPARATE SUBROUTINE SUB ( X )
      REAL, INTENT(IN) :: X            REAL, INTENT(IN) :: X
    SEPARATE                         END SUBROUTINE SUB
  END TYPE T

CONTAINS
  COMPLETING SUBROUTINE SUB ! Finishing SUB defined in T
    ... ! Body of SUB
  END SUBROUTINE SUB
END MODULE M
```

4.5 Procedure interface before CONTAINS and body after CONTAINS

```
MODULE M
PRIVATE

PUBLIC :: SUB
SUBROUTINE :: SUB ( X )   or     SEPARATE SUBROUTINE SUB ( X )
  REAL, INTENT(IN) :: X            REAL, INTENT(IN) :: X
SEPARATE                         END SUBROUTINE SUB

! Note to other humans:  All of the public procedures of this module
! are written as incomplete procedures above this point.  For
! purposes of knowing the interface to this module, there's no need
! to read beyond this point.

CONTAINS
  COMPLETING SUBROUTINE SUB ! Finishing SUB defined before CONTAINS
    ... ! Body of SUB
  END SUBROUTINE SUB
END MODULE M
```