Date: April 2, 1997
To: X3J3
From: William B. Clodius
Subject: Example Syntaxes for Parametric Procedures and Modules

I. Introduction

This paper presents a relatively well developed example syntax for parameterization in Fortran. For completeness, parameterization of modules, derived types, and procedures are all considered, although parameterization of modules and derived types are similar in their capabilities. The current proposal is based on an extension of Java by Odersky and Wadler, 1997, although several other languages have similar facilities, see the references at the end. It differs from the current parameterized derived types proposal in providing parameterization in terms of types in general, and not just indirectly through kind values, and in providing a special "signature" construct used to provide an abstract definition of the types used in parameterization. Because of the importance of abstract types for object orientation, the implications of signatures for object oriented Fortran are also discussed where appropriate. In addition to the simple examples in the main part of the paper, more extensive examples are given in an appendix.

II. Type "signature" definition

The main limitation of the current parameterized derived types proposal is its restriction to parameterization by integers. General parameterization requires parameterization in terms of types, which in turn benefits from a means of specifying the characteristics of those types that can be used as parameters.

Such a specification is most clearly given by a construct that defines an abstract type in terms of its name, the names and types of its public components, and the names and abstract definitions of a set of procedures or operators that have dummy arguments or return values with the signature name as their types. For an object oriented language, this specification might also indicate whether the type is monomorphic or polymorphic or is related to a specific type through inheritance. In addition to specifying which types can be used as parameters, it also provides a concrete syntax for specifying abstract polymorphic "classes", should Fortran become object oriented.

The literature provides several terms for such an abstract type definition, but the two most common terms are type interface or type signature. Therefore, either the INTERFACE construct should be extended so that it can provide an abstract definition of data types as well as procedural types, or a new SIGNATURE construct should be provided as a means of defining abstract data types. Such a definition should include the type signature name, a type definition construct specifying public components, and the pertinent interface constructs.

Example:

```
   SIGNATURE :: ordered

      TYPE :: ordered
         PRIVATE
      END TYPE ordered

      INTERFACE OPERATOR (<)
         FUNCTION LESS_THAN(X, Y)
            LOGICAL :: LESS_THAN
            TYPE(ordered), INTENT(IN) :: X, Y
         END FUNCTION LESS_THAN
      END INTERFACE OPERATOR (<)

   END SIGNATURE ordered
```

The signature of ordered specifies an abstract type with no pertinent public components and one defined operation, <. Such a type can be useful in defining generic sorting procedures. While the above provides an explicit syntax for defining an abstract type, it might be useful to specify signatures implicitly by example, or default, i.e., if Fortran implements inheritance.

III. Type “signature” association

The type signature has meaning only when associated with one or more specific types. The relationship of the signature to a type could be specified in at least three different ways: as a separate construct, as part of the signature construct, or as part of the derived type construct. Each of these has different uses. For completeness all uses are given below although only one use has direct application to parameterization.

A. Association as a separate construct

Specifying the relationship as a separate construct has two applications. First, it can be used to specify that an argument for a parameterized construct represents a type with the desired signature. Ideally this should have a syntax similar to that of the type declaration statement.

Example:

```
   SIGNATURE(ordered) :: X
```

might indicate that the argument X represents a type with the signature ordered, and all occurrences of X within the module will be replaced by the actual argument type, if compatible, upon parameterization. Second, should Fortran become object oriented, it can be used to indicate that the signature represents a dynamic type and type X is intended to be one form of that type.

Example:

```
   TAGGED(ordered) :: X
```

B. Association as part of a signature construct

Specifying the relationship as part of a signature construct could be used to constrain applicability of the signature to a fixed ordered set of types. Such a fixed ordered set is similar to Fortran's intrinsic types with their different KINDs. A natural syntax is then to use KIND as a keyword in such a specification.

Example:

```
   SIGNATURE :: intrinsic_ordered

      KIND :: CHARACTER, INTEGER(KIND=1), INTEGER(KIND=2), &
         INTEGER(KIND=4), REAL(KIND=4), REAL(KIND=8)
      TYPE :: intrinsic_ordered
         PRIVATE
      END TYPE intrinsic_ordered

      INTERFACE OPERATOR (<)
         FUNCTION LESS_THAN(X, Y)
            LOGICAL :: LESS_THAN
            TYPE(intrinsic_ordered), INTENT(IN) :: X, Y
         END FUNCTION LESS_THAN
      END INTERFACE OPERATOR (<)

   END SIGNATURE intrinsic_ordered
```

which defines an abstract type with six representations, all of them intrinsic types. Any reference to TYPE(intrinsic_ordered(KIND=1)), would then refer to the default character type, TYPE(intrinsic_ordered(KIND=2)), would refer to the intrinsic INTEGER with kind value 1 (usually, but not always, a BYTE), etc. It appears to be straight forward to provide the capabilities of the intrinsic kind selectors for user defined type signatures.

C. Association as part of the derived type construct

Specifying the relationship as part of the derived type construct could be used to provide an independent specification of a type facilitating independent compilation

Example:

```
TYPE ORDERED_SET
   IMPLEMENTS ordered
   PRIVATE
   INTEGER, ALLOCATABLE :: Component
END TYPE ORDERED_SET
```

IV. A syntax for parameterizing derived types

A. Type definition statement

A straightforward syntax for parameterizing derived types, based on the current parameterized derived types proposal, would be to add an optional list of module parameter names in parens following the type-name in the type-definition statement.

Example:

```
TYPE matrix(sig, dim)
```

Unlike the current parameterized derived types proposal, the dummy arguments can be type signatures as well as integer values. The actual type should be indicated by an explicit SIGNATURE or INTEGER declaration in the derived type definition.

Example:

```
TYPE matrix(sig, dim)
   SIGNATURE(number) :: sig
   INTEGER :: dim
   TYPE(sig) :: element( dim, dim)
END TYPE matrix
```

Note number in this case is a previously defined signature that might be compatible not only with type REAL, as in the example in the parameterized derived types proposal, but also with type COMPLEX, or special derived types such as the interval arithmetic proposal.

B. Entity declaration

1. Simple declaration syntax

The obvious syntax for parameterized derived type instantiation would follow that of the parameterized intrinsic types. The type parameter values would therefore be specified in parens after the type name, in either keyword or positional form.

Integer arguments for a kind type parameter shall be an integer initialization expression. The expression for a non-kind type parameter may be either a specification expression or assumed. The most straightforward syntax for signature parameters restricts the arguments to type specifiers. The resulting syntax is fairly straightforward

Examples:

```
TYPE(matrix(REAL, 1000)) :: a
TYPE(matrix(sig=COMPLEX(KIND=4), dim=1000)) :: b
TYPE(matrix(TYPE(INTERVAL), 1000)) :: c
```

2. Sophisticated declaration syntax

The concept of type in other languages includes such concepts as arrays and pointers. Further, the addition of the elemental attribute in Fortran 95 means that if an entity of a given type will satisfy a signature then an array or pointer of that type will often satisfy that signature. A full generalization of this parameterization capability would allow the specification of selected attribute specifiers as well as the type specifiers. This generality could be achieved by letting the signature arguments be a type specifier followed by the pertinent attribute specifiers. Such a combination must be either textually separated, i.e., by parens or an appropriate constructor, e.g. SIGNATURE

Examples:

```
TYPE(matrix((REAL, DIMENSION(2,2)), 1000)) :: d
TYPE(matrix(sig=SIGNATURE(COMPLEX(KIND=4), POINTER(2,2)), &
   dim=1000)) :: e
```

V. A syntax for parameterizing modules

A. Module definition statement

Because module parameterization and type parameterization are similar in effect it is not clear that the language requires both, but for completeness both will be discussed. Much of the syntax and semantics of parameterized modules follows from that of the parameterized derived types. A parameterized module can be specified by adding an optional list of module parameter names in parens following the module-name in the module-definition statement.

Example:

```
MODULE matrix(sig, dim)
```

B. Module declaration

The restrictions on the parameters are essentially identical to those discussed above for parameterized derived types. The instantiation of a module could be either on its use

Example:

```
   USE matrix(REAL, 1000)
```

or in the definition of a new module

Example:

```
   MODULE complex_matrix = matrix(COMPLEX(KIND=4), 1000)
```

Both instantiation syntaxes for parameterized modules are liable to be less frequently used then the instantiation syntax for parameterized derived types. Therefore there is likely to be generated less duplicate code generated by unsophisticated implementations for parameterized modules than for types. The second form of a module declaration, by providing a specific name for the module, is less likely to require significant name mangling which can complicate interfacing to code from other processors.

VI. A syntax for parameterizing procedures

A. Parameterized procedure definition

The most obvious syntax for parameterized procedures is to allow functions to return procedures as values or provide a special construct, e.g. FUNCTOR. Issues in choosing a new procedure type include: Would users have trouble understanding a function returning functions? Should the returned procedure be defined statically? Would users expect that a function could be used dynamically? Should access to global variables be restricted for such procedures? Would such restrictions be expected of functions?

Examples:

```
   ! Find the element with a maximum value in the one dimensional
   ! array a of type sig which can be ordered using the relational
   ! operator, <.
   FUNCTION max_element( sig )
      SIGNATURE(ordered) :: sig
      FUNCTION max_element( a)
         TYPE(sig) :: a(:)
         TYPE(sig) :: max_a
         TYPE(sig) :: max_element(2)
         INTEGER :: size_a, i
         size_a = SIZE(a)
         max_element = a(1)
         DO i=2, SIZE(a)
            IF ( max_element < a(i)  ) max_element = a(i)
         END DO
      END FUNCTION max_element
   END FUNCTION max_element

   FUNCTOR max element( sig )
```

```
   SIGNATURE(ordered) :: sig
   FUNCTION max_element( a)
      TYPE(sig) :: a(:)
      TYPE(sig) :: max_a
      TYPE(sig) :: max_element(2)
      INTEGER :: size_a, i
      size_a = SIZE(a)
      max_element = a(1)
      DO i=2, SIZE(a)
         IF ( max_element < a(i)  ) max_element = a(i)
      END DO
   END FUNCTION max_element
END FUNCTOR max_element

! Valid is .TRUE. if c represents a digit in radix r.
! Count returns the number of times radix has been executed
FUNCTION radix(r)
   INTEGER, INTENT(IN) :: r
   INTEGER :: n = 0
   SUBROUTINE radix(c, valid, count)
      CHARACTER, INTENT(IN) :: c
      LOGICAL, INTENT(OUT) :: valid
      INTEGER, INTENT(OUT) :: count
      n = n + 1
      count = n
      SELECT CASE (c)
         CASE (‘0’:’9’)
            valid = ( IACHAR(c) - IACHAR(‘0’) ) < r
         CASE (‘a’:’z’)
            valid = ( 10 + IACHAR(c) - IACHAR(‘a’) ) < r
         CASE (‘A’:’Z’)
            valid = ( 10 + IACHAR(c) - IACHAR(‘A’) ) < r
         CASE DEFAULT
            valid = .FALSE.
      END SELECT
   END SUBROUTINE radix
END FUNCTION radix

FUNCTOR radix(r)
   INTEGER, INTENT(IN) :: r
   INTEGER :: n = 0
   SUBROUTINE radix(c, valid, count)
      CHARACTER, INTENT(IN) :: c
      LOGICAL, INTENT(OUT) :: valid
      INTEGER, INTENT(OUT) :: count
      n = n + 1
      count = n
      SELECT CASE (c)
         CASE (‘0’:’9’)
            valid = ( IACHAR(c) - IACHAR(‘0’) ) < r
         CASE (‘a’:’z’)
            valid = ( 10 + IACHAR(c) - IACHAR(‘a’) ) < r
         CASE (‘A’:’Z’)
            valid = ( 10 + IACHAR(c) - IACHAR(‘A’) ) < r
```

```
            CASE DEFAULT
               valid = .FALSE.
         END SELECT
      END SUBROUTINE radix
   END FUNCTOR radix
```

B. Procedure Declaration

The arguments to parameterized procedures should be restricted to initialization expressions and signatures. The current procedure pointer proposal introduces a syntax for procedure pointers

Example:

```
   ABSTRACT INTERFACE

      SUBROUTINE sub(x,y)
         REAL :: x,y
      END SUBROUTINE

      FUNCTION fun(x) RESULT(f)
         REAL :: x,f
      END FUNCTION

   END INTERFACE

   PROCEDURE(fun), POINTER :: a => NULL(), b, c
```

that could be readily extended to such declarations.

Examples:

```
   ABSTRACT INTERFACE

      SUBROUTINE sub2(c, valid, count)
         CHARACTER, INTENT(IN) :: c
         LOGICAL, INTENT(OUT) :: valid
         INTEGER, INTENT(OUT) :: count
      END SUBROUTINE

      FUNCTION fun2(x) RESULT(f)
         LOGICAL :: f
         REAL :: x(:)
      END FUNCTION

   END INTERFACE

   PROCEDURE(sub2) :: radix_hex = radix(16)
   PROCEDURE(fun2) :: max_real = max_element(REAL)
```

which could then be used as

```
      CALL radix hex( char, valid, count)
```

```
WRITE(*,*) char,' is a hexadecimal character? ',valid, &
   ' determined on call ',count,' of radix'

WRITE(*,*) max_real(a),' is the maximum element of array a.'
```

References

Gerald Baumgartner and Vincent F. Russo, "Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++," Software--Practice & Experience, 25 (8), pp. 863-889, August 1995. (Implemented in GNU C++)

Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers, "Subtypes vs. Where Clauses: Constraining Parametric Polymorphism," OOPSLA'95 Conference Proceedings, Pages 156-158, ACM Press, October 1995. (Discusses the implementation in Theta)

M. P. Jones, "A system of constructor classes: overloading and implicit higher-order polymorphism," Proc. Functional Programming Languages and Computer Architecture, pages 52-61, ACM Press, June 1993. (Discusses the basis of polymorphism in the functional language Haskell.)

Andrew C. Myers, Joseph A. Bank, and Barbara Liskov, "Parameterized Types for Java," Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages®, Paris, France, January 15-17, 1997, Pages 132-145, ACM Press. (Based on Theta's implementation discussed by Day et.al.)

Martin Odersky and Philip Wadler, "Pizza into Java: Translating theory into practice," Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages®, Paris, France, January 15-17, 1997, Pages 146-159, ACM Press.

Appendix A: Additional Examples

The following gives additional examples of parameterization in the syntax suggested by the main part of this paper. The examples are intended to both illustrate the syntax and the power of parameterization.

1. Generic Stack data type module

This and the following example are based on the Ada code given in Sebesta, p. 420 -421, and 426, which defines a module (package) which implements a generic stack data type. The first implementation uses parameterized modules, the second parameterized types. Both examples are provided to illustrate how comparable their capabilities are, and the differences in syntax of the two implementations. A close Fortran equivalent of the above would be

```
   MODULE generic_stack(sig, max_size)
      PUBLIC

      SIGNATURE :: element_type
! Element_type has no built-in operations
         TYPE element_type
            PRIVATE
         END TYPE element_type
      END SIGNATURE element_type

      SIGNATURE(element_type) :: sig
      INTEGER :: max_size ! A generic parameter for stack size

! The duplication on stacktype in the signature might not
! be necessary
      TYPE stacktype
         TYPE(sig) :: list(max_size)
         INTEGER :: topsub=0
      END TYPE stacktype

      SIGNATURE :: stacktype

         TYPE stacktype
            TYPE(sig) :: list(max_size)
            INTEGER :: topsub=0
         END TYPE stacktype

         INTERFACE
            FUNCTION empty (stk)
               LOGICAL :: empty
               TYPE(stacktype), INTENT(IN) :: stk
            END FUNCTION empty
            SUBROUTINE push (stk, element)
               TYPE(stacktype), INTENT(IN OUT) :: stk
               TYPE(element_type), INTENT(IN) :: element
            END SUBROUTINE push
            SUBROUTINE pop (stk)
               TYPE(stacktype), INTENT(IN OUT) :: stk
            END SUBROUTINE pop
            FUNCTION top (stk)
               TYPE(ELEMENT_TYPE) :: top
               TYPE(stacktype), INTENT(IN) :: stk
            END FUNCTION top
         END INTERFACE
      END SIGNATURE

   CONTAINS
```

```
      FUNCTION empty (stk)
         LOGICAL :: empty
         TYPE(stacktype), INTENT(IN) :: stk
         empty = (stk%topsub == 0)
      END FUNCTION empty

      SUBROUTINE push (stk, element)
         TYPE(stacktype), INTENT(IN OUT) :: stk
         TYPE(element_type), INTENT(IN) :: element
         IF (stk%topsub >= max_size) THEN
            WRITE(*,*) "ERROR - Stack overflow"
         ELSE
            stk%topsub = stk%topsub + 1
            stk%list(stk%topsub) = element
         END IF
      END SUBROUTINE push

      SUBROUTINE pop (stk))
         TYPE(stacktype), INTENT(IN OUT) :: stk
         IF (stk%topsub == 0) THEN
            WRITE (*,*) "ERROR - Stack underflow"
         ELSE
            stk%topsub = stk%topsub - 1
         END IF
      END SUBROUTINE pop

      FUNCTION top (stk)
         TYPE(ELEMENT_TYPE) :: top
         TYPE(stacktype), INTENT(IN) :: stk
         IF (stk%topsub == 0) THEN
            write (*,*) "ERROR - Stack is empty"
         ELSE
            top = stk%list(stk%topsub)
         END IF
      END FUNCTION top
   END MODULE generic_stack
```

which could be instantiated with the statements

```
   USE generic_stack(max_size=100,sig=INTEGER), $
         generic_stack => integer_stack
```

or

```
   MODULE STACK_OF_REAL_VECTORS = $
      generic_stack(max_size=100,sig=(REAL,DIMENSION(10)))
```

2. Generic Stack data type

This example implements a stack data type using parameterized derived types. In the following, the entity declaration form, TYPE(stacktype(sig, max_size)) ..., is used although it is wordier than that, TYPE(stacktype) ..., used in the parameterized module syntax. The wordier form is used to maintain similarity with the current proposed parameterized derived type syntax, although the simpler entity declaration syntax of the parameterized modules, appears to be also usable with parameterized types.

```
   MODULE generic_stack
      PUBLIC

      SIGNATURE :: element_type
! Element_type has no built-in operations
         TYPE element_type
            PRIVATE
         END TYPE element_type
      END SIGNATURE element_type

      TYPE stacktype(sig, max_size)
         SIGNATURE(element_type) :: sig
         INTEGER :: max_size ! A generic parameter for stack size
         TYPE(sig) :: list(max_size)
         INTEGER :: topsub=0
      END TYPE stacktype

      SIGNATURE :: stacktype

         TYPE stacktype(sig, max_size)
            SIGNATURE(element_type) :: sig
            INTEGER :: max_size
            TYPE(sig) :: list(max_size)
            INTEGER :: topsub=0
         END TYPE stacktype

         INTERFACE
            FUNCTION empty (stk)
               LOGICAL :: empty
               TYPE(stacktype), INTENT(IN) :: stk
            END FUNCTION empty
            SUBROUTINE push (stk, element)
               TYPE(stacktype), INTENT(IN OUT) :: stk
               TYPE(element_type), INTENT(IN) :: element
            END SUBROUTINE push
            SUBROUTINE pop (stk)
               TYPE(stacktype), INTENT(IN OUT) :: stk
            END SUBROUTINE pop
            FUNCTION top (stk)
               TYPE(ELEMENT_TYPE) :: top
               TYPE(stacktype), INTENT(IN) :: stk
            END FUNCTION top
         END INTERFACE
      END SIGNATURE

   CONTAINS

      FUNCTION empty (stk)
         LOGICAL :: empty
```

```
      TYPE(stacktype(sig, max_size)), INTENT(IN) :: stk
      empty = (stk%topsub == 0)
   END FUNCTION empty

   SUBROUTINE push (stk, element)
      TYPE(stacktype(sig, max_size)), INTENT(IN OUT) :: stk
      TYPE(element_type), INTENT(IN) :: element
      IF (stk%topsub >= max_size) THEN
         WRITE(*,*) "ERROR - Stack overflow"
      ELSE
         stk%topsub = stk%topsub + 1
         stk%list(stk%topsub) = element
      END IF
   END SUBROUTINE push

   SUBROUTINE pop (stk))
      TYPE(stacktype(sig, max_size)), INTENT(IN OUT) :: stk
      IF (stk%topsub == 0) THEN
         WRITE (*,*) "ERROR - Stack underflow"
      ELSE
         stk%topsub = stk%topsub - 1
      END IF
   END SUBROUTINE pop

   FUNCTION top (stk)
      TYPE(ELEMENT_TYPE) :: top
      TYPE(stacktype(sig, max_size)), INTENT(IN) :: stk
      IF (stk%topsub == 0) THEN
         write (*,*) "ERROR - Stack is empty"
      ELSE
         top = stk%list(stk%topsub)
      END IF
   END FUNCTION top
END MODULE generic_stack
```

which could be instantiated with the statements

```
TYPE(generic_stack(max_size=100,sig=INTEGER)) :: &
   STACK_OF_INTEGERS

TYPE(generic_stack(max_size=100,sig=(REAL,DIMENSION(10))) :: &
   STACK_OF_REAL_VECTORS
```

A possible problem with this instantiation syntax is that it requires name mangling and hence complicates interfacing to non-Fortran code. This might not be a problem with a more fully developed C language interface, or it could be addressed by allowing an alternate type declaration syntax

```
TYPE INTEGER_STACK= generic_stack(max_size=100,sig=INTEGER)

TYPE(INTEGER_STACK) :: STACK_OF_INTEGERS
```

```
   TYPE REAL_VECTOR_STACK = &
      generic_stack(max_size=100,sig=(REAL,DIMENSION(10))
   TYPE(REAL_VECTOR_STACK) :: STACK_OF_REAL_VECTORS
```

3. A generic sorting procedure

The following example is is based on the Ada code given in Sebesta, p. 355, and implements a generic sorting procedure for the elements of a vector

```
   FUNCTOR generic_sort( sig )
      SIGNATURE(ordered) :: sig
      SUBROUTINE generic_sort(list)
         TYPE(sig), INTENT(IN OUT) :: list(:)
         TYPE(sig) :: temp
         INTEGER :: list_size
         intrinsic :: shape
         list_size = shape(list)
         DO index_1= 1, list_size-1
            DO index_2 = index_1+1, list_size
               IF (LIST(index_2) < list(index_1)) THEN
                  temp := list(index_1)
                  list(index_1) = list(index_1)
                  list(index_2) = temp
               END IF
            END DO
         END DO
      END SUBROUTINE generic_sort
   END FUNCTOR generic_sort
```

which could be instantiated by

```
   ABSTRACT INTERFACE

      SUBROUTINE sub3(list)
         INTEGER, INTENT(IN OUT) :: list(:)
      END SUBROUTINE

   END INTERFACE

   PROCEDURE(sub3) :: integer_sort = generic_sort(INTEGER)
```

4. A generic vector to scalar procedure

Fortran 90's SUM and PRODUCT can be thought of as functions which take an array as their arguments and return a scalar that is the result of recursively applying a binary function with its first argument that is the first element of the array and the second argument the result of applying the function to the rest of the array. It is sometimes useful to define other functions which take an array as their arguments and return a scalar that is the result of such a recursive application of a binary function. The following, based on the ML function “reduce” of Ullman, pp. 104-105, defines a FUNCTOR generalizing this capability

```
   FUNCTOR vector_to_scalar( binary_function)
      SIGNATURE sig
         TYPE sig
            PRIVATE
         END TYPE sig
         FUNCTION binary_function( a, b)
            TYPE(sig), INTENT(IN) :: a, b
            TYPE(sig) :: binary_function
         END FUNCTION binary_function
      END SIGNATURE sig
      FUNCTION binary_function( a, b) ! Redundant
         TYPE(sig), INTENT(IN) :: a, b
         TYPE(sig) :: binary_function
      END FUNCTION binary_function
      FUNCTION vector_to_scalar( a) RESULT (scalar)
         TYPE(sig) :: a(:), scalar
         IF ( SIZE(a) == 1 ) THEN
            scalar = a
         ELSE
            scalar = binary_function( a(1), &
                        vector_to_scalar(a(2:)) )
         END IF
         RETURN
         END
      END FUNCTION vector_to_scalar
   END FUNCTOR vector_to_scalar
```

which could be instantiated by

```
   ABSTRACT INTERFACE

      FUNCTION func4(list)
         TYPE(interval), INTENT(IN) :: list(:)
         TYPE(interval) :: func4
      END SUBROUTINE

   END INTERFACE

   PROCEDURE(func4) :: sum_interval = !
      vector_to_scalar(plus_interval)
```

5. A compositional function

It is often useful to define a function that is the result of applying two functions in succession to a value, e.g., H(x) = F(G(X)), this can be provided by hand coding in detail this function in Fortran 90, but it can be useful to have a shorthand for this definition. The following FUNCTOR, based on the ML function “comp” of Ullman, pp. 108-110, provides such a shorthand

```
FUNCTOR composition( f, g)
   SIGNATURE sig1
      TYPE sig1
         PRIVATE
      END TYPE sig1
   END SIGNATURE sig1
   SIGNATURE sig2
      TYPE sig2
         PRIVATE
      END TYPE sig2
   END SIGNATURE sig2
   SIGNATURE sig3
      TYPE sig3
         PRIVATE
      END TYPE sig3
   END SIGNATURE sig3
   INTERFACE
      FUNCTION f(x)
         TYPE(sig1) :: x
         TYPE(sig2) :: f
      END FUNCTION f
      FUNCTION g(y)
         TYPE(sig3) :: y
         TYPE(sig1) :: g
      END FUNCTION g
   END INTERFACE
   FUNCTION composition (z)
      TYPE(sig3) :: z
      TYPE(sig2) :: composition
      composition = f( g( z ) )
   END FUNCTION composition
END FUNCTOR composition
```

which could be instantiated as

```
ABSTRACT INTERFACE

   FUNCTION func5(z)
      REAL, INTENT(IN) :: z
      REAL :: func5
   END SUBROUTINE

END INTERFACE

PROCEDURE(func5) :: coscos = composition(cos, cos)

PROCEDURE(func5) :: sincos = composition(sin, cos)

PROCEDURE(func5) :: expcos = composition(exp, cos)
```

instead of hand coding

```
FUNCTION coscos(x)
   REAL :: x, coscos
```

```
      coscos = cos( cos(x))
   END FUNCTION coscos

   FUNCTION sincos(x)
      REAL :: x, sincos
      sincos = sin( cos(x))
   END FUNCTION sincos

   FUNCTION expcos(x)
      REAL :: x, expcos
      coscos = exp( cos(x))
   END FUNCTION expcos
```

References

Robert W. Sebesta, “Concepts of Programming Languages, Third Ed.,” Addison-Wesley, Reading, Mass., 1996.

Jeffrey D. Ullman, “Elements of ML Programming,” Prentice-Hall, Englewood Cliffs, New Jersey, 1994.