Feature Proposals for Fortran
– A subset

by Espen Myklebust

Contents

1 Access specification in USE statements 2
2 Read-only variables 2
3 Extensions to IMPORT statements 4
4 Read-only components in derived types 6
5 Extension to the IF statement 7
6 Additional operator symbols 9
7 New intrinsic procedures 9
8 Extensions to intrinsic procedures 11
9 Initialization 12
10 Extension to the INTENT attribute 16
11 Extension to assignment statements 17
12 Object pseudo-components 18
13 Qualification of temporary objects 18
14 New syntax for iterated DO loops 20
15 Array of pointers to scalar entities 22
16 Implicit compatibility inheritance for derived types 24
1 Access specification in USE statements

1.1 Proposal

Possibility to prevent inheritance of USE associations

1.2 Rationale

- At present one of the following strategies is necessary to achieve the desired behavior
  - Declare the MODULE to be PRIVATE and explicitly list all the entities in the MODULE in a PUBLIC statement
    * The preferred method, but may be cumbersome if there are many entities to list
  - Declare all entities accessible by USE association in a PRIVATE statement
    * Awkward as all entities from the USEd MODULE must be known unless an ONLY clause is used, but that may be just as awkward if there are many entities that are needed

- Very useful when there are a lot of entities in both the host MODULE and the USEd MODULE which should all be public

1.3 Proposed syntax

\[
\text{USE } [[, \text{module-nature}] [,, \text{access-spec}] ::] \text{ module-name } [, \text{ONLY: only-list}] \\
\text{USE } [[, \text{module-nature}] [,, \text{access-spec}] ::] \text{ module-name } [, \text{rename-list}] 
\]

where \text{access-spec} is either PUBLIC or PRIVATE

1.4 Comments

- A PUBLIC attribute is default, just as for the MODULE itself
- PROTECTED could in principle also be allowed and would make PUBLIC data PROTECTED in scopes using the module in which this was specified
  - Affect access to shared data only
  - This very “special” functionality might have limited practical application

2 Read-only variables

In this section a \textit{variable} is understood as an object that is stored in memory and, hence, has a memory address, without necessarily being mutable (in a particular scope). The reason is that the word \textit{constant} typically refers to a Fortran \texttt{PARAMETER} which, at least conceptually, does not have a memory address but is “hard coded” into the program.

A similar feature proposal can be found at fortranwiki.org, and was proposed there by JOE KRAHN.
2.1 Proposal

Add an attribute to flag variables as read-only (immutable), but, in contrast to to parameters, with a requirement that they be addressable to allow them as pointees. Also add a statement with the same token to declare host- or USE-associated variables as read-only in the scope of the statement.

2.2 Rationale

- A variable declared with the read-only attribute would be allowed in initialization expressions
  - Equivalent to parameters
- Read-only variables would be valid as pointer targets
  - The pointer could possibly be required to have the same attribute to help protecting the read-only target
  - This is disallowed for an entity with the parameter attribute as it is not (required to be) addressable
- A non-pointer dummy argument may not have the read-only attribute
  - INTENT(IN) provides the same functionality
  - combination with VALUE would be meaningless
- Applied to a pointer, it would prevent assignment of a value since a read-only object is not allowed on the lhs. of an assignment statement
  - This can be very useful even if the target is not read-only
    * Data protection analogous to INTENT(IN) in procedures
  - The pointer should then be allowed to have a (USE-associated) target with access specification PROTECTED
  - Allocation of the pointer would only be meaningful if a source is provided since no (ordinary) assignments would be allowed
  - For dummy arguments the attribute would pertain to the target value and, hence, be orthogonal to how the intent attribute applies to pointers
    * The attribute would “cancel out” the pointer attribute if combined with INTENT(IN)
      - This specific attribute combination should still be allowed since it would make changes to the INTENT straightforward without affecting the target protection
- The corresponding statement would make it possible to
  - have data available through USE association behaving as PROTECTED in one scope and as PUBLIC in another (as long it actually is PUBLIC)
  - protect host associated variables in internal procedures and block constructs from being altered
- Makes it possible for the programmer to have large structures as read-only variables without risking (as this is processor dependent) that the executable becomes excessively large (if it contains the whole structure), while still maintaining the possibility to have “hard coded” data objects through usage of parameters.
2.3 Proposed syntaxes

2.3.1 Attribute name

- **CONSTANT**
  - Used in the following

- **PROTECTED**
  - Can cause confusion since a read-only variable may or may not be a **PROTECTED** variable from a **MODULE**

2.4 Examples

```fortran
REAL, CONSTANT :: const1 = 4.0, const2 = 5.0
REAL :: var1 = 7.3
REAL, CONSTANT, POINTER :: ptr

ptr => const1
ptr = 3.5 !Not allowed
ptr => const2
ptr => var1
ptr = 3.5 !Also not allowed
```

2.5 Comments

- A **CONSTANT** variable would always have a memory address
  - A **TARGET** attribute would thus not alter the behavior, and could be implied (and hence omitted)

- Used as attribute for local variables in procedures the semantics could be different from that of **PARAMETERS** in being “reinitializable” at every entry using (possibly non-scalar) specification expressions

3 Extensions to **IMPORT** statements

3.1 Renaming

3.1.1 Proposal

Allow an **import-rename-list** in **IMPORT** statements as an alternative to the **import-name-list**.
3.1.2 Rationale

- If different names are used for an entity in the implementation and in the host which the entity is imported from, one have to rename the imported entities in the interface after copying the subprogram header from the code of the implementation.
- A renaming facility minimizes the work by writing each name association only once.
- Adds clearness by working as a “conversion table” so that when looking at the code of the procedure implementation it is transparent which names are renamed and which ones are the same.

3.1.3 Proposed syntax

```plaintext
INTERFACE
  IMPORT :: implemented-entity-name => host-entity-name
  interface-body
END INTERFACE
```

3.2 Restrict the available host objects

3.2.1 Proposal

Allow `IMPORT` statements in internal and `MODULE` procedures and `BLOCK` constructs to restrict which named entities (or possibly only data objects) that are accessible. The default behavior (i.e. without an `IMPORT` statement) would be as at present, implying an `IMPORT` statement without an `import-name-list` (or an `import-rename-list`).

3.2.2 Rationale

- Data hiding is an important way of simplifying writing (and compiling?) of programs.
- This would be an alternative to having an external or module procedure and a unit calling it sharing (a restricted set of) entities through a `MODULE`.
  - When different procedures need access to different entities a similar functionality can be quite complex to achieve through `USE` associations.

3.2.3 Comment

- It could be a desirable behavior that an `IMPORT`ed variable with either the `ALLOCATABLE` or `POINTER` attribute would “lose” these, but that they can be re-obtained by naming the variable in an `ALLOCATABLE` or `POINTER` statement.
  - Similar to the case that the variable was provided as an actual argument to the (possibly non-internal) procedure without these attributes being explicitly provided in the declaration of the corresponding dummy argument.
  - To hinder for instance unintentional respecification of bounds on assignment.
  - This behavior could possibly (?) be achieved by using the variable as an `selector` in an associate construct.
4 READ-ONLY COMPONENTS IN DERIVED TYPES

* The *associate name* would necessarily differ from the original variable name which would make the host association lose some of its convenience
  · When desired, the above proposed renaming feature will furnish this functionality

3.2.4 Additional proposal

Allow the import statement to take the form

```
IMPORT NONE
```

This would provide the possibility to have non-host-associated procedures with explicit interfaces without the need for a separate `MODULE` to “wrap” the procedure. This will lead to increased security and less writing when no host associations are needed and it is judged that a `MODULE` is not strictly necessary.

3.2.4.1 Comment

- This would be the default behavior for `INTERFACE` blocks

4 Read-only components in derived types

4.1 Proposal

A method to make read-only access possible for a component in a derived type

4.2 Rationale

- At present “get and set” methods must be used in conjunction with a `PRIVATE` component attribute to obtain a read-only behavior
  - The “set” method would have to be `PRIVATE` in the scope(s) in which the component should have read-only access

4.3 An example

```
TYPE [[, type-attribute-list] ::] type-name [(type-parameter-name-list)]
  :
  type-name, PROTECTED :: component-declaration-list
  :
END TYPE [type-name]
```

4.4 Comment

- Basically adding the `PROTECTED` attribute to the presently allowed `access-spec` in `component-attribute-spec-list`
5 Extension to the IF statement

5.1 Proposal

Extend the IF statement to allow for specification of evaluation order of the logical expressions. Alternatively, new logical operators that guarantee short-circuit evaluation of AND and OR can be added to provide an equivalent (and more general) mechanism.

5.2 Rationale

- Having the possibility to perform short-circuit evaluation in IF constructs will at times significantly simplify the code, and also make it more clear
  - It is not possible to reach an ELSE block and/or ELSEIF block(s) if one must use nested IF constructs to ensure that evaluation of the logical expressions are done in the correct order
- Short-circuit operators, although more general and flexible, might be of limited importance outside IF constructs
  - In this case the extension to the IF statement may be a better solution

5.3 Proposed syntaxes

5.3.1 Alternative IF statement syntax

\[ \text{[ELSE]} \text{IF (logical-expr1) [op-token (logical-expr2)...]} \text{ THEN} \]

where op-token is either AND or OR.

5.3.1.1 Comments

- op-token is not an operator per se, rather a part of the IF statement, and should therefore not have the dots on each side
- A natural extension to the present IF statement in that op-token, in the case of AND, merely represents “THEN; IF”
- The same extension may be allowed for DO WHILE loop statements:
  DO WHILE (logical-expr1) [op-token (logical-expr2)...]

5.3.2 Additional operators

\[ \text{logical-expr1 .ANDTHEN. logical-expr2} \]
\[ \text{logical-expr1 .ORELSE. logical-expr2} \]
5.3.2.1 Comment

- The proposed operators should be available for general logical expressions (not limited to those in IF statements)
  - How useful this will be is questionable and therefore the extended IF statement is preferred here.

5.4 Examples with IF constructs

In the below block of code it is necessary to add an extra IF construct when \textit{expr2} can only be evaluated if \textit{expr1} returns \texttt{.TRUE.}.

```plaintext
lvar = .FALSE.
IF (logical-expr1) THEN
  IF (logical-expr2) THEN
    lvar = .TRUE.
  END IF
END IF
IF (lvar) THEN
  :
ELSE
  :
```

This could however, using the proposed syntax, be vastly simplified into

```plaintext
IF (logical-expr1) AND (logical-expr1) THEN
  :
ELSE
  :
```

Similarly, in the case that the \textit{op-token} above was \texttt{OR} it would be equivalent to the following block of code:

```plaintext
lvar = .FALSE.
IF (logical-expr1) THEN
  lvar = .TRUE.
ELSEIF (logical-expr2) THEN
  lvar = .TRUE.
END IF
IF (lvar) THEN
  :
ELSE
  :
```
6 Additional operator symbols

6.1 Proposal

Extend the set of possible operator symbols that can be used for user-defined operators to include some, or all, of the F03 special characters. Allowing composites/aggregates of the added characters and the defined operator symbols should be considered.

6.2 Rationale

- This significantly enhances the expressiveness of the language.

6.3 Comments

- At least the following special characters should be considered: \ ~ ^
- Would it be possible to include the colon (:) as a symbol for using in an operator definition?
  - It has no other use in the execution section of a program (?)

7 New intrinsic procedures

7.1 A PRINT function

7.1.1 Proposal

Add a function for converting scalar numeric variables to character representation.

7.1.2 Rationale

- At present this must be done through a WRITE statement which can be somewhat inconvenient when building a character string by concatenation
  - Must write numbers to a temporary character variable and concatenate sequentially using the TRIM function
- Such a function should return the shortest possible string containing the number to be converted, optionally using a format specifier
  - Equivalent to if TRIM was applied to the string

7.1.3 Proposed syntax

\[
\text{PRINT}(\text{numeric-expr} \ [, \text{fmt-spec}])
\]

where \text{numeric-expr} is a scalar expression of numeric type, and \text{fmt-spec} is an ordinary Fortran format descriptor which may be an asterisk or, equivalently, may be omitted. A field width in the descriptor is always ignored and is allowed to be and asterisk or a question mark, and the same goes for a repeat count as only a single numeric data entity is allowed.
7.2 A function for testing sign

7.2.1 Proposal

Add an elemental function for testing the sign bit of (intrinsic) numeric types, returning a logical value corresponding to the sign

7.2.2 Rationale

- Often it is necessary to know the sign of a value (examples should be unnecessary), and so a simple function to test the sign bit of a numeric value should be available
  - It would be faster and more explanatory than comparing to zero
  - Checking the sign bit explicitly is possible using \texttt{BTEST}, but not portable as the interpretation of bits is processor dependent ([2])
    * The sign bit of integers are typically the last bit, requiring the slightly awkward syntax \texttt{BTEST(I,BIT\_SIZE(I)-1)}
    * If this is to be done with \texttt{REALs}, the \texttt{TRANSFER} intrinsic must be used (?) producing even more cryptic code
- For \texttt{COMPLEX} the result should be a 2 element rank 1 array corresponding to the sign of the real and imaginary components
  - If the sign of only one component is desired that is easily achievable through pseudo-component qualification
7.2.3 Proposed syntax

- \texttt{TEST}(X)
  - Resembles \texttt{BTEST}
  - Short, but the meaning may not be obvious
  - Used in the following
- \texttt{TEST\_SIGN}(X)
  - More explanatory
  - Somewhat lengthy

7.2.4 Proposed return value

- \texttt{.TRUE.} if the value is negative (i.e. sign bit is set) and \texttt{.FALSE.} otherwise
  - Same as if \texttt{BTEST} was used for integers
  - \texttt{TEST}(0) will return \texttt{.FALSE.} which is common in languages where numeric values can be used interchangeably with logical values (if existent)
    * Signed zeros will return the values according to their signs

7.2.5 Examples

In an iteration using the logarithm for computing the relative difference between two consecutive values

\begin{verbatim}
DO WHILE ( \texttt{TEST(var\_1) .EQV. TEST(var\_2)} ) AND ( \texttt{LOG(var\_1/var\_2) .GT. tol} ) :
\end{verbatim}

where the syntax proposed in section 5 has been used. To test if two complex numbers are in the same quadrant would amount to

\begin{verbatim}
\texttt{ALL( TEST(Z1) \texttt{.EQV. TEST(Z2)} )}
\end{verbatim}

8 Extensions to intrinsic procedures

8.1 Extend the \texttt{FINDLOC} function

8.1.1 Proposal

Allow a \texttt{POINTER} argument as an alternative to \texttt{VALUE}. The function result should be the vector of subscript positions identifying the element associated with the \texttt{POINTER}. The argument associated with the \texttt{POINTER} argument must be a scalar pointer.
8.1.2 Rationale

- When a (scalar) pointer is associated with an array element there is no way of easily determining the indices of the element besides testing the status of `ASSOCIATED` for each array element
  - an intrinsic implementation could be much more efficient since it’s probably possible to go “the opposite way” because the memory addresses of the `POINTER` and the array elements are known to the program

- The `ASSOCIATED` function cannot be used in testing for the whole array due to the semantics of that function

8.1.2.1 Comment

- It may perhaps seem odd to have this situation in the first place, but it occurs for instance when having two-way connectivity between elements (objects) and nodes (objects) in a finite element program

9 Initialization

Below are some proposals regarding the initialization of values to variables in procedures. There might be a more far-reaching meaning attached to the word “initialize” when used in relation with Fortran, possibly regarding if it is possible to accomplish at compile-time or not. Here, though, it is used in the meaning “the first/initial value” a variable has when referenced in the executable part of a (sub-) program.

9.1 Dummy arguments

9.1.1 Proposal

Allow initialization for dummy arguments with `INTENT(OUT)`, and for any `OPTIONAL` argument with the effect that a missing argument is initialized to the assigned value. The behavior will differ from other (explicit) initializations by not having an implied `SAVE` attribute.

9.1.2 Rationale

- As initialization is not presently allowed for dummy arguments the proposed feature will not break any existing (standard conforming) code

- Dummy arguments with `INTENT(OUT)` are often used to hold diagnostic information, such as a status indicator, which usually have some default value in the sense that if nothing exceptionally happens it should assume this value
  - At present this value must be assigned through an ordinary assignment which may happen anywhere in the execution part
  - Performing this initiation to a default value (not to be confused with default initiation!) in the declaration makes it clear which value that is
An **OPTIONAL** argument can be used to add some parameter to a procedure that otherwise take a default value, or to hold diagnostic information (cf. e.g. with the **STATUS** argument of many intrinsic **SUBROUTINES**)

- Frequently there exists a local variable that will be assigned the value of the optional argument, in the case the latter is present
- If it was possible to assign the **OPTIONAL** argument a default value when missing, this “duplicate” local variable is not needed as the dummy argument would always be defined

- In neither of the two cases an implied **SAVE** attribute makes any sense
- For an argument with both **OPTIONAL** and **INTENT(OUT)** specified the dummy is initialized even if it is associated with an actual argument
  - as if the **INTENT(OUT)** behavior takes precedence over **OPTIONAL**

### 9.1.3 Comment

- When a dummy argument has a default value as proposed, it should be allowed in a **restricted** (also called **general**) **specification expression**

### 9.2 Regarding implied **SAVE**

The present rules about (explicit) initialization of variables and the implied **SAVE** attribute, as well as default initialization, creates considerable confusion among many Fortran users (and not only the newbies!), which is apparent from the many lengthy Internet forum discussions. Presented below are two alternatives to help sort this out.

#### 9.2.1 Alternative 1

Add a constraint in the standard that compilers must be able to issue a warning or error (depending on the option/switch) for any initialized variable (that implicitly acquires the **SAVE** attribute) that do not have the **SAVE** attribute. It must be considered good programming practice to explicitly declare all variables that are saved with the **SAVE** attribute, especially to assist (code) readers not familiar with all fine details of the language. This would be the same behavior as for default initialized module variables at present (at least up to F03).

#### 9.2.2 Alternative 2

Add a constraint in the standard that compiles must have an option to choose whether or **not** to add an implicit **SAVE** attribute to non-module variables, so that this attribute must be added explicitly to obtain the present behavior. This is the more radical alternative, but should be the preferred as it is the most logical behavior (judging by most forum posts from users unaware of the present behavior) as the **SAVE** attribute actually exists in the language. As the present behavior would still be available through the proposed option, all existing code would still be compilable, and it could be combined with another option to issue a warning when unsaved variables are initialized in the declaration in order to help the transition for users that are accustomed to the now standard behavior.
9.2.2.1 Comments

- The F08 behavior that all module data objects (implicitly?) have the `SAVE` attribute seems logical since a `MODULE` is not called in the sense that a subprogram is.
  - As a mental construct, a `MODULE` could be considered as existing from the program initiation, but only accessible when being `USED`; while a subprogram (at least a `RECURSIVE` one) is “instantiated” each time it is called.

9.2.3 Additional proposal to alternative 2

If `SAVE` is not implied by initialization one could allow restricted expressions (termed general specification expressions in [2]) to be used for value initializations

- A larger class of variables can then be initialized without using an assignment expression
  - Fewer variables will be in a “undefined” state
- Encourages a programming style where variables that is initializable by only input arguments and possibly (valid) functions of such ones, instead of deferring the initialization to the point where the variable is actually needed
  - Makes it easier to understand a program
  - Deferring the initialization may also be error prone
    * If the arguments used for computing it is not declared with `INTENT(IN)`
    * If it is forgotten all together so that the value is `undefined`
    * A programming language should discourage this
  - It makes it clear which values belong to the “problem definition” of the subprogram
    * Analogous to how a mathematical problem is stated, e.g. initial value problems and boundary value problems
- This would allow (among several useful things) more straight forward usage of automatic arrays (as an alternative to `ALLOCATABLE` ones) when the extents are given by expressions that depend on the value(s) of already declared variables

```
SUBROUTINE sub(a,b)
    ...
    INTEGER :: c = F(a,b), d = G(a,c), e = H(b,c)
    REAL, DIMENSION(d-c,d+e) :: array
    ...
END SUBROUTINE sub
```

- It makes the relation between the variables and the array extents apparent, in addition to being much more compact than having to initiate and allocate the local variables in the execution part
- If, in addition, c, d and e were allowed to be declared with the `CONSTANT` attribute (cf. comment in section 2) they would serve merely as place holders for the expressions
- The legacy behavior, with only initialization expressions allowed for value initialization, would be regained if the variable was declared with the `SAVE` attribute
9.3 Automatic arrays

9.3.1 Proposal & rationale

Allow scalar initialization expressions (or scalar restricted expressions if the previous proposal in this section is approved) to be used to initialize automatic arrays since the expression would be compatible with the variable regardless of the array extents. The same would probably be possible for default initialization of array components in derived types with extents given by \texttt{LEN} type parameters.

9.3.2 Example

FUNCTION foo(a,b)
  INTEGER, INTENT(IN) :: a,b
  REAL, DIMENSION(a,b) :: c = 1.
END FUNCTION foo

9.3.3 Additional proposal 1

Allow a \textsc{MOLD} argument in the \texttt{DIMENSION} attribute of an automatic array instead of explicit specification of the array extents, similar to how allocatable arrays may be \texttt{ALLOCATED} using a \textsc{MOLD} argument. The array used as \textsc{MOLD} must be a local or an explicit or non-\texttt{OPTIONAL} assumed shape dummy argument array that was declared before. Functionality similar to the \texttt{SOURCE} argument of \texttt{ALLOCATE} would be to specify the initial value of the automatic array to be equal to the array given as the \textsc{MOLD} argument. Example:

FUNCTION foo(arr)
  INTEGER, DIMENSION(:,,:), INTENT(IN) :: arr
  REAL, DIMENSION([MOLD =] arr) :: temp = arr
END FUNCTION foo

9.3.3.1 Comments

- If an automatic object is initialized using the with the same array as specified as the \textsc{MOLD} object, it could be allowed to use an asterisk instead since it would be redundant
  - This is analogous to the F08 syntax for \texttt{PARAMETERs}

9.3.4 Additional proposal 2

Require that compilers have (at least) an option to allocate automatic arrays in the pool/heap/? if they are too large for the stack. Put differently: require safe memory allocation for automatic arrays.
10 Extension to the INTENT attribute

This feature proposal is more or less “snipped” from the Feature Proposals page at fortranwiki.org, and was proposed there by the user MAX.

10.1 Proposal

Allow INTENT(NONE) to signal a dummy dummy-argument.

10.2 Rationale

- This sounds silly at first, but it is already used in some intrinsic procedures, such as the MOLD argument of the TRANSFER statement
- Generally, it is useful for aiding generic-to-specific procedure mapping, as in the TRANSFER example
- It could also be used to flag intentionally-unused arguments, which are occasionally useful
  - This is for instance the case when commenting lines of code, i.e. not using some variables, for debugging purpose, while not wishing all the verbose of the compiler saying “this or that has been declared but not used”

10.3 Additional extension

In the case that specification expressions are allowed in value initializations (see section 9.2.3) dummy arguments with INTENT(NONE) could be allowed used in the specification part still, and usage would be prohibited only in the execution part.

10.3.1 Comment

- This might not actually be an additional extension, as it is analogous to how the MOLD argument is used in the TRANSFER function, but rather an illustration of how the extension could be used.

10.3.2 Example

Often it is desired to use a specific (typically higher than default) precision and/or range in the internals of a procedure, such as when numeric iterations are performed or when intermediate results are expected to be much larger than the final result.

```fortran
FUNCTION func(a,b,c)
  REAL(lp) :: func
  REAL(lp), INTENT(NONE) :: a,b  !low precision
  INTEGER,INTENT(IN) :: c       !c > 0 ^ |LOG(c)| >> 0
  REAL(hp) :: a_hp = a          !or REAL(a,hp)
  REAL(hp) :: b_hp = b          !or REAL(b,hp)

  func = (a_hp**c + b_hp**c)**(1./c)
END FUNCTION func
```
11 Extension to assignment statements

11.1 Proposal

Allow a special character to be used in place of the designator for the variable on the l.h.s. of an assignment statement in the expression on the r.h.s.

11.2 Rationale

- Explicates that one of the arguments/primaries is being changed
- Eases optimization (in obvious ways)
  - These optimizations are hopefully done in all major Fortran compilers, but it may still be useful as a crutch for those whose analysis capabilities are very limited
- When used with “reallocation on assignment” it could be required that the any pointers to the object was kept intact
- If the variable designator is long it saves space and enhances clearness
- Makes it less error prone to duplicate expressions where the assignment variable designator is a sub-object and the sub-object qualifiers differ among the expressions

11.3 Proposed character

- ~
  - Used for the exact same purpose by BURROUGHS for ALGOL
  - Should be reserved to be used as an operator symbol (cf. with section 6)
- @
  - Reference to the variable at the l.h.s. of the assignment sign
  - The symbol is not used in Fortran and is not easily confused with any operator

11.3.1 Examples

Listed below is three assignments with plausibly increasing difficulty of optimization:

\[
\begin{align*}
\text{idx} &= @ + 5 \\
\text{arr}(3:idx) &= @*2 \\
\text{arr} &= [@, -@]
\end{align*}
\]

from a trivial case (I hope!), via an array operation on an array section, to a reallocation where, when the proposed syntax is used, it should be obvious to the compiler that an optimization should be attempted.

The following lengthy assignments
would be written simply as

\[
\begin{align*}
\text{array}(i,j)\%\text{comp1}(k,l)\%\text{value1} &= \text{ABS}(\text{array}(i,j)\%\text{comp1}(k,l)\%\text{value1} & - \text{SQRT}(\text{array}(i,j)\%\text{comp1}(k,l)\%\text{value1})) \\
\text{array}(i,k)\%\text{comp1}(j,l)\%\text{value2} &= \text{ABS}(\text{array}(i,k)\%\text{comp1}(j,l)\%\text{value2} & - \text{SQRT}(\text{array}(i,k)\%\text{comp1}(j,l)\%\text{value2}))
\end{align*}
\]

The chance of producing a typo if the second statement is made by copying the first statement and just altering the subscripts, is practically eliminated.

### 12 Object pseudo-components

Allow intrinsic inquiry functions to be accessed through pseudo-component syntax when no optional arguments are needed (along the same lines as the F08 \%re and \%im pseudo components for COMPLEX variables)

#### 12.1 Array pseudo-components

- array\%ubound
- array\%shape
- array\%allocated

#### 12.2 String pseudo-components

- string\%len
- string\%len_trim

### 13 Qualification of temporary objects

#### 13.1 Function-result qualification

#### 13.1.1 Proposal

Allow referencing an array element or a type component from a function result without storing it to a temporary variable first.
13.1 Function-result qualification

13.1.2 Rationale

- According to [1] “An array section, a function reference, or an array expression in parentheses must not be qualified by a subscript list.”
  - and likewise for a component designation in the latter two cases?
- This leads to the need for temporary variables that may not be needed for anything besides storing the result in order to allow access to an array element or type component
  - For various reasons such temporary variables should be allocatable and so would need to be deallocated manually (if they are large) after use unless they go out of scope
  - Alternatively an ASSOCIATE block can be used, and this is probably a better alternative, but the syntax is still elaborate if only a single reference is needed
- To automatically let the compiler (i.e. the “system”) deal with this should be straightforward as the temporary objects are created and destroyed automatically as needed

13.1.3 Proposed syntax

```
array_valued_fun(arg-list) | [subscript-list]
array_valued_fun(arg-list) | (subscript-list)
derived_type_valued_fun(arg-list) | % component-spec
```

where the pipe character is used to “pipe the result” entity to a (hidden) temporary variable that can be qualified as usual.

13.1.3.1 Comment

- In the first line the proposed new array qualifier syntax is used

13.1.4 Example

If a procedure returns a derived type which has a bound function a function composition can be performed which is not otherwise possible without using explicit temporary objects. For instance:

```
c = fun1(arg-list1) | % tb-fun2(arg-list2) | % tb-fun3(arg-list3)
```

would be equivalent to the following code:

```
a = fun1(arg-list1)
b = a | % tb-fun2(arg-list2)
c = b | % tb-fun3(arg-list3)
```
13.2 Array constructor qualification

13.2.1 Proposal & rationale

- In the same manner as for function results it should be possible to use the same syntax for array constructors
- Effectively allow so-called linear indexing, i.e. using a single subscript to access an array object
  - even more versatile since it is not limited to a single, whole array
- For relatively simple constructors and subscript there is no need to make a temporary array copy
  - Some of the functionality can be obtained using storage association, but
    * arbitrary data objects cannot be combined,
    * it is limited to static variables, i.e. excludes ALLOCATABLEs, function results and constants, and
    * is generally advised against

13.2.2 Proposed syntax

\[
[ \text{ac-value-list} ] | [ \text{subscript-list} ]
\]
\[
[ \text{ac-value-list} ] | \% \text{component-spec}
\]

14 New syntax for iterated DO loops

14.1 Proposal

Add a new syntax for declaring a DO statement loop-control that follows the pattern of modern Fortran. This syntax will be further enhanced with subsequent syntax proposals

14.2 Rationale

- The “regular” DO construct index/control specification has a completely different form than the modern Fortran constructs
  - A comma separated list specifying the control variable (instead of a triplet)
  - a label (legacy from non-block DO)
  - no parenthesized expression or specification
- It also lacks some additional features of the DO CONCURRENT and FORALL index specifications
- A consistent notation should be pursued
  - extremely helpful when learning the language
14.3 Proposed syntax

\[\text{[do-construct-name:] \ DO \ [,] \ FOR( [type-spec ::] \ index-spec-list \&} \newline
\[\text{[\&, scalar-mask-expr] )} \newline
\text{\textbf{:\ \}}} \newline
\text{END \ DO \ [do-construct-name]}\]

This matches the \texttt{DO CONCURRENT} syntax exactly, with the obvious difference in the “DO-keyword.” However, if more than one index variable is specified there must be a rule for determining in which order the indices are incremented, in contrast to the case for a \texttt{DO CONCURRENT} construct where this does not matter. The following rule is suggested:

The leftmost index declared is incremented first, then the second leftmost and so on.

This is consistent with how the \textit{subscript order value} increases in \textit{array element order} which have a pleasant implication when looping through arrays.

14.3.1 Example

The following loop

\begin{verbatim}
DO j=1,m
  DO i=1,n
    IF ( i.LT.n .AND. j.LT.m ) THEN
      a[i,j] = a[i,j] + a[i+1,j+1]
        END IF
  END DO
END DO
\end{verbatim}

could be written much more compactly as

\begin{verbatim}
DO FOR( i=1:n, j=1:m, i.LT.n .AND. j.LT.m )
  a[i,j] = a[i,j] + a[i+1,j+1]
END DO
\end{verbatim}

In both loops the indices are specified in the order that should(?) achieve the best performance, but whereas for the present syntax the outermost construct (usually declared \textit{first}?) must have the \textit{last} index and the innermost construct the first one, the proposed syntax declares the indices in the same order as they (should!) appear in the array qualifier. This is perhaps a minor detail (that some compilers automatically mend), but in the writers experience, at least beginners and those that are not aware of this, tend to declare the nested loops “in the reverse order.” Of course it would also be possible to maintain the structure of the first code block, only changing the loop-control parts to the new syntax, as shown below

\begin{verbatim}
DO FOR( j=1:m )
  DO FOR( i=1:n )
    : 
\end{verbatim}
14.3.2 Comments

- The proposed syntax makes it easy to switch between a CONCURRENT and regular DO by just changing the token, as opposed to having to respecify the whole statement.
- Compared to the present “legacy” DO construct it also adds the possibility
  - to declare multiple index/control variables,
  - their type parameter, and
  - for a mask expression.
- The token FOR is widely used for this kind of loop.
- In the last code block it could be possible to include the masking expression since the loop control variables would be equally well known to the program as in the middle code block.

14.4 Additional extensions

14.4.1 Modern implied DO

The modern triplet syntax could also be used in an implied DO specification, with the obvious candidate syntax

\[
\text{do-value-object-list, do-variable = int-expr:int-expr [:int-expr] }
\]

where the do-value-object-list can be another implied DO specification or a list of values or objects (with the same rules as for the present types of implied DO specifications).

15 Array of pointers to scalar entities

15.1 Proposal

Add an attribute to declare that a pointer’s target(s) is (are) scalar. Specifically, in the case that a DIMENSION or RANK attribute is given, it is associated with the pointer structure itself.

15.2 Rationale

- Often it is desired to have an array of pointers pointing to scalar entities, as opposed to an array pointer pointing to an array (subsection)
  - For example individual elements of an array that are impossible to designate using a subscript triplet
- At present this can be done by creating a derived type containing a POINTER component and the declare an array of this type.
  - The dummy derived type that has to be made complicates the code unnecessary, and can be confusing
15.3 Proposed syntax

15.3.1 Attribute name

- **SCALAR**
  - An established Fortran token (it is also used as a keyword for in the inquiry function `ALLOCATED`)
  - Used in the following

- **SCALARTARGET**
  - More clear about the meaning, but somewhat lengthy
    * The meaning should be clear enough with only `SCALAR` since a `POINTER` necessarily has a `TARGET`
  - The literal meaning of the word implies a single target which has to be interpreted as to pertain to each element of an array

- **SCALARPOINTER**
  - Should replace the `POINTER` attribute and any of the above all together
  - Signals in a clear way that this is not the standard `POINTER` (by replacing it)

15.3.2 Examples of declarations

A special attribute list, the `scal-attr-list`, is used in the following examples which differs from any other valid attribute list only in that it cannot include `DIMENSION` (or `RANK`). This is intentional to emphasize the the array cases which are the important ones here.

\[
\text{type-spec, [scal-attr-list,], POINTER, SCALAR :: variable-declaration-list}
\]

which is no different than the present scalar pointer declaration apart from the `SCALAR` attribute which is allowed for consistency. Next the array cases:

\[
\text{type-spec, [scal-attr-list,], DIMENSION(colon-list), POINTER, SCALAR :: var-decl-list}
\]
\[
\text{type-spec, [scal-attr-list,], RANK(rank), POINTER, SCALAR :: var-decl-list}
\]

Specifically, the `scal-attr-list` may include an `ALLOCATABLE` attribute, which naturally pertains to the pointer structure.
15.4 Examples

```fortran
TYPE :: real_ptr
    REAL, POINTER :: num
END TYPE real_ptr

REAL :: a
TYPE(real_ptr), DIMENSION(n) :: real_ptr_arr
REAL, DIMENSION(n), POINTER, SCALAR :: real_ptrs

real_ptr_arr(1)%num => a
real_ptrs(1) => a
```

The `num` component lengthens the statement and it seems artificial when what is desired is just an array of (pointers to) REALs. Likewise, if the derived type POINTER component was a derived type, the actual TARGET component would have to be preceded by the actual POINTER component name, and this is easily forgotten when the “path” becomes long (which, almost ironically, may be the result of the very issue addressed here).

15.5 Comments

- When a POINTER is declared with both the SCALAR and DIMENSION (or RANK) attribute, it cannot have the CONTIGUOUS attribute
- When a derived type object has scalar data and/or procedure POINTER components, if this object is array-valued a reference to such components will be an array of scalar pointers

16 Implicit compatibility inheritance for derived types

16.1 Proposal

An attribute to declare a component of a derived type to hold the “main data.” The derived type should (implicitly) inherit all applicable/compatible (and available) procedures and operators, both intrinsic and non-intrinsic, from this component. For a composite derived type, the remaining components could be considered, in a broad sense, as “meta data.”

This particular component will be referred to as the data-component from here on.

16.2 Rationale

- Often it is desirable to group data and meta data together in a composite type; or
- to make special cases of intrinsic arrays (e.g. rank 1 arrays of extent 3 for spatial vectors) which is used frequently.
  - It is also an appealing way to gather data objects to make arrays of such types, instead of creating higher rank and/or extent arrays.
To operate on the *data-component* one must use component syntax, i.e. `variable%component`, but that is neither efficient (when programming), nor does it have an appealing look

- the alternative is then to write type-bound procedures for all needed procedures and operators, which can be a considerable amount of work

- The type can be used anywhere the *data-component* otherwise could be used

- The result of a usage of the type where it is not otherwise applicable, but where the *data-component* is, is just the same as if the *data-component* was used specifically through component syntax

16.2.1 Comments

- The component can be of any type
  - intrinsic or derived
  - scalar or array

- There can be no more than one *data-component* in a type definition

- It could be beneficial in an implementation to store an array of a type containing a *data-component* in a separate (parallel) array to optimize performance

16.3 Proposed syntax

16.3.1 Attribute name

- **DATA**
  - Preferred as it underlines the component’s role of the component holding the *Data*

- **PASS**
  - To signal that the particular component is *passed*, implicitly, to any procedure (including those associated with an operator) not specifically written for the type.
  - May cause confusion with the PASS attribute of a type-bound procedure in that the latter passes the *whole entity*, which may well be a composite

16.3.2 An example

```fortran
TYPE [[,type-attribute-list] ::] type-name [(type-parameter-name-list)]
  :
  type-name, DATA :: component-declaration-list
  :
END TYPE [type-name]
```
16.4 Examples of use

TYPE :: sqmat(n)
  INTEGER, LEN :: n
  REAL, DIMENSION(n,n), DATA :: mat
  REAL, DIMENSION(n) :: eigvals
  TYPE(vec(n)), DIMENSION(n) :: eigvecs
:END TYPE sqmat

All intrinsic array procedures and operators, and in addition all available non-intrinsic with argument(s) compatible with the mat component, will be applicable to an instance of the sqmat type. One might of course define type-bound procedures in addition or as substitutes for one or more of the intrinsic ones as usual.

TYPE(sqmat(3)) :: sqmat3
REAL, DIMENSION(3,3) :: a3by3array
sqmat3 = a3by3array*sqmat3
sqmat3 = SIN(sqmat3)

sqmat3%mat is first multiplied (in the standard element-wise manner) with a3by3array, and the resulting 3-by-3 array is assigned back to sqmat3%mat. The result is the same as if the mat component would be explicitly specified each time sqmat3 is used. Then the intrinsic ELEMENTAL function SIN is applied to all the elements of sqmat3%mat which is then assign back to the same array. Furthermore, the following two statements are equivalent and they will both print a scalar which is the sum of all the elements of sqmat3%mat

PRINT *, SUM(sqmat3)
PRINT *, SUM(sqmat3%mat)

16.4.1 Comments

- The main thrust of this feature proposal is the inheritance of procedure and operator compatibility, not the “syntactic sugar” of omitting the reference to (what is here called) the data-component explicitly
  - Although there is just a principal difference between the two notions here, the difference will be essential if extended to arrays, as discussed in [4]

- Specifically, the ASSIGNMENT(=) operator is, if it is defined for the data-component, implicitly inherited from it, which is very convenient as it is not necessary to use a structure constructor as long as a specific ASSIGNMENT procedure is not defined
  - It should also be possible to use a derived type with a compatible DATA component in an assignment statement, even when the types themselves are not the same
References


