Subject: Alternative to .ANDTHEN. and conditional expressions
From: Van Snyder
References: 04-385, 04-390, 04-393, interp 00031/00074

1 Number

TBD

2 Title

Alternative to .ANDTHEN. and conditional expressions,

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide for actual arguments that are not evaluated until the called procedure requests it.

6 Rationale

If a procedure could decide when to evaluate the actual arguments associated with its dummy arguments, it would be possible for users to develop the equivalent of the .ANDTHEN. and conditional expressions proposed in 04-390 and 04-393 and their predecessors. This proposal is more general. Implementing .ANDTHEN. and conditional expressions using this proposal is almost certainly less efficient than would be the case if they were intrinsic, but in cases where users' subprograms delay evaluating (perhaps thereby not evaluating) complicated arguments, especially ones involving function references, it could have implications for improved performance. This also provides yet another way to get extra information, of which a "library" procedure is unaware, from the caller of the library procedure into code the library procedure needs to execute.

7 Estimated Impact

Minor for the standard. Needs a new dummy argument attribute and a new statement.

8 Detailed Specification

Provide an attribute for dummy arguments that specifies that the corresponding actual argument is not evaluated until the called procedure requests it to be evaluated. Provide a statement that causes one or more dummy arguments to be evaluated. When an actual argument is evaluated, it is evaluated in the context of the instance of the procedure or main program wherein it appeared as an actual argument. This instance may be different from the instance current as the moment of evaluation if the actual argument first appeared as an argument within a recursive procedure. The values of variables used when the argument is evaluated are those current as of the moment the argument is evaluated, which may be different from the values current as of the moment the argument first appeared as an actual argument. The processor is not expected to determine automatically when to evaluate the argument. It is the program's responsibility explicitly to cause the argument to be evaluated. Indeed, the program may cause the argument to be evaluated several times. It is a program error if the argument is referenced, including as an actual argument associated with a dummy argument that does not defer evaluation, before it is evaluated.

Evaluating an actual argument that is a variable consists of evaluating all expressions within it, not taking a copy of it.

The attribute might be spelt LAZY or DEFERRED of DELAYED. The statement might be of the form "EVALUATE (*dummy-arg-name-list*)".

A processor that uses copy-in/copy-out instead of by-reference argument association will necessarily need to copy-out every non-lazy non-intent(in) argument before evaluating a lazy argument.

8.1 Examples

```
logical function ANDTHEN ( A, B )
  logical, intent(in) :: A
  logical, intent(in), lazy :: B
  andthen = a
  if ( .not. a ) return
  evaluate ( b )
  andthen = b
end function ANDTHEN

logical function ORELSE ( A, B )
  logical, intent(in) :: A
  logical, intent(in), lazy :: B
  orelse = a
  if ( a ) return
  evaluate ( b )
  orelse = b
end function ORELSE

type(T) function IF ( A, B, C )
  pointer :: IF
  logical, intent(in) :: A
  type(t), intent(in), lazy, target :: B
  type(t), intent(in), lazy, target, optional :: C
  if ( a ) then
    evaluate ( b )
    if => b
    return
  end if
  if ( .not. present(c) ) then
    nullify ( if )
    return
  end if
  evaluate ( c )
  if => c
end function IF
```

For the IF function to be useful as an implementation of conditional expressions with the semantics described in 04-393, it is necessary that a reference to a function with a pointer result be allowed as an actual argument associated with a nonpointer dummy argument with intent other than INTENT(IN) (the opposite to the currently-proposed outcome of interp 00031/00074, which has twice failed letter ballots), and that a disassociated pointer actual argument that is argument associated with an optional nonpointer dummy argument be considered to be absent instead of an error, as described in 04-385.

Example:

```
call sub1 ( if(present(a),a(:,i)) ) ! Slice of A, else absent
  ! Unfortunately, A needs the TARGET attribute in the calling scope
```

A disadvantage of a nonintrinsic IF is that users need to write a separate one for each type and kind.

Lazy evaluation provides yet another way to get non-state information into user-provided code for library routines. Consider a reference to a nonlinear solver:

```
  call solver ( state, myFunc(state,extraStuff), myDeriv(state,extraStuff) )
```

Assuming the dummy arguments corresponding to the references to the `myFunc` and `myDeriv` functions are lazy, the solver can evaluate them as many times as it needs, each time in the context of the caller. They are *not* dummy procedures, and `extraStuff` does not go through `solver`, not even as a component of an extensible type. If the results of `myFunc` and `myDeriv` are allocatable, `solver` would need to be carefully written so as not to leak memory: the dummy arguments would need to be allocatable, and `solver` would need to take care to deallocate them, since the automatic deallocation of allocatable function results wouldn't happen until `solver` finally returns, which might not be until after several evaluations of `myFunc` and `myDeriv`.

```
  subroutine Solver ( State, Func, Deriv )
    real, intent(inout) :: State(:)
    real, intent(in), lazy, allocatable :: Func(:), Deriv(:,:)
    ...
    do
      ...
      if ( allocated(func) ) deallocate( func )
      evaluate ( func )
      ...
      if ( allocated(deriv) ) deallocate ( deriv )
      evaluate ( deriv )
      ...
    end do
    ...
  end subroutine Solver
```

9 History