****

J3/02-315

Date: November 2002
To: J3
From: Aleksandar Donev
Subject: ELEMENTAL Procedures and Efficiency, and PURE
Reference: J3-007R3

****

Summary

I point to some problems with efficiently using ELEMENTAL procedures in Fortran 2002 and suggest a compiling scheme that would fix this, along with some needed backward-compatible modifications to the current draft that would make this implementation more feasible. This was partly inspired by recent posts/discussions with Van Snyder, as well as past personal frustrations with elementals in Fortran 95. Problems also exist with PUREs, discussed later.

NOTE: The extended example is not in this printout.

ELEMENTAL

Required changes to the way ELEMENTALs work include (in order of priority):
1. Remove the constraint prohibiting elementals as actual arguments and procedure pointer targets, and,
2. Making sure that non-intrinsic elementals and non-elementals are not mixable, i.e., an elemental procedure pointer, type-bound procedure or procedure dummy argument should only be allowed to point, be bound to, or be argument associated with a non-elemental.
3. Add the dummy argument attribute SCALAR (and possibly NONSCALAR) that guarantee that the actual will be scalar.
Either
4a. Allow INTENT(OUT) SCALAR arguments, or even better,
4b. Remove the purity requirement on elementals and replace with the condition that order of execution be unimportant to the user.

The problem described is part of a much more general issue of writing TKR-generic procedures in Fortran (think templates in C++ or generic classes in Eiffel), but no attempt is made here to solve the general problem (it is coming shortly, after consultation with Van and others). Besides, it is too late to finally introduce genericity properly in Fortran. But it is not too late to fix elementals!

The proposed implementation is only a suggestion to vendors. I am aware that it will not be high on their priority list. But at least we are making the way for such efficient implementations of elementals later.

PURE

PURE procedures are another vastly underused F95 feature. They are a very nice concept, but the basic problem is that the restrictions on PURE are too strong. For example, one cannot even raise an error flag (think exceptions), or even point a pointer to some global data (I am not saying modify the data, but just point to the data). This is especially true because ELEMENTALs are required to be PURE. The real requirement is that the order of execution of ELEMENTALs not matter.

I make suggestions here on how to relax this for ELEMENTALs, but do not try to fix PURE ones.

The Problem

There is a frequently appearing task in Fortran programming which still finds no good solution: Making procedures that can be called with arrays of arbitrary rank...efficiently. Assume for the moment that these routines can be written, using array-syntax, in a way that is otherwise independent of the rank. Elementals provide this, albeit only for pure operations that completely disregard the possible "arrayness" of the arguments. More importantly, there is an efficiency problem with elementals which has unfortunately made them a very seldomly used array feature in Fortran 95 (with the exception of, of course, the intrinsics).

The problem with elementals is that to really have efficiency, in a lot of cases, the loop over the array elements needs to go *inside* the elemental procedure, rather then outside a procedure call to a scalar version of the elemental. I will assume that inlining is not possible, and in the examples I give this is the case, either because dynamic binding is used with OO approaches, or because the user actually provides the elemental procedure. Vendors typically implement the loop around the call, possibly since this is easier. However, with a few modifications in the way ELEMENTAL procedures work, a few deletions of restrictions on elementals which make no sense, and a change in attitude from compiler writers, the usefulness of ELEMENTALs can significantly be increased.

This issue is compounded with OO approaches, especially because of dynamic dispatch, whose overhead can dramatically be reduced if the slot-table lookup is done externally to the loop. Examples of the need for rank-indepedendent procedures include random number generation (see comments on non-purity below), as well as a whole collection of problems in which the user provides a routine which operates on scalar arguments but in reallity needs to be called on a grid of points {I attach at the end a (long) example of numerical quadrature of a one-dimensional function Int[f(x),x=a..b]}.

A noticeable feature in both these examples, and in general a stumbling block to putting the loop in the elemental, is the fact that some of the arguments, typically the passed object in OOP solutions, are scalar. A commonly occuring feature is also that the user can predict this ahead of time and would be better off restricting this to be a scalar indeed. I suggest that a new attribute for dummy arguments of ELEMENTAL procedures be added, say SCALAR, which would state that this argument is not part of the "elementalness" of the procedure.

I would really like to design a novel set of language features that deals with this issue of genericity, i.e. writing code which applies to different TKR actual arguments without any change. Kind of like templates in C++, but with the added complexity of the rank. However, this is a major job. Therefore, I resort to elementals, and suggest some modifications which would make them a viable replacement.

The Solution

I give an example using random number generation below. Random number generation is not pure, and the ordering of the generation does matter for the final result. However, for the user the ordering does not matter (otherwise he/she should not call the generator with an array argument but put the look around the scalar version him/herself), just like with the RANDOM_NUMBER intrinsic. I have assumed below that the restriction on SCALAR arguments being intent(in) has been removed, although this issue is of course separate from this proposal. For now, just take this as a very simple, yet illustrative, example.

Here is how the declaration of an OO random number generator would look like in my proposed scheme:

```
module Random_Number_Generation

   type, extensible :: random_number_generator
      integer :: seed=0
   contains
      procedure(next_number), pass(generator), deferred :: next_number
         ! Generates the next number in the pseudorandom stream
   end type random_number_generator
```

```
   abstract interface
      elemental subroutine next_number(generator,number)
         ! Can be used to generate arrays of random numbers
         class(random_number_generator), SCALAR, intent(inout) :: generator
         real, intent(out) :: number
      end function next_number
   end abstract interface

end module Random_Number_Generation
```

For users that really care about efficiency, the above generator, without the SCALAR, would not be a satisfactory solution in F2x, due to the above efficiency concerns. The solution one would have to resort to, is a very ugly one, in which there is a generic binding for next_number, which has to overriden in every specific extension (implementation) of a random-number generator. So now every time we extend random_number_generator we have to write 8 (or maybe 9, with one special case for contiguous arrays) different specific procedures, which will inline the random-number generator manually. This produces long, obscure, macro-monsters of codes which sadly I have to write almost every day. Notice that we cannot write these rank-specific routines only once, and then reuse them in subsequent extensions, since we do not yet have access to the scalar random-number generator at this API stage.

If instead, the compiler actually generated these 8/9 versions of the elemental procedure for arguments of different rank automatically (which is very simple), simplicity and efficiency would be combined in a very appealing way. So an elemental procedure will in fact become something like a dope-vector with 8/9 procedure pointers, one for scalars, one for ranks 1 through 7, and one for the case when all arrays involved are contiguous. Elemental dummy procedure arguments, procedure pointers, and type-bound elementals will in fact be implemented as such dope-vectors. Of course this is just a suggestion and there other alternatives. It should be up to the vendors to choose a strategy, and consistently use it in all cases elementals are involved.

Some will raise the issue of what to do when some of the actual arguments for one of the non-SCALAR arguments are also scalar (as they can be according to the current rules for ELEMENTAL). In this case, the old strategy of simply putting the loop around the scalar version will be satisfactory, and is still available to the compiler. Or the compiler can simply replicate the scalar to an array, depending on what it chooses to do. We could also add another dummy argument attribute, say NONSCALAR, which will guarantee that the actual is an array.

Required Changes and Edits

I do ask for some help to ensure that these are complete. I state the required functionality first, and then the essential edits:

1. Remove the constraint prohibiting elementals as actual arguments and procedure pointer targets, but make sure that
2. Non-intrinsic elementals and non-elementals should not be mixable, i.e., an elemental procedure pointer, type-bound procedure or procedure dummy argument should only be allowed to point, be bound to, or be argument associated with a non-elemental.

Edits:

Procedure pointers:
143:28 Delete C728 and possibly replace with:
C728: If <proc-target> is a nonintrinsic elemental procedure, then <proc-pointer-object> shall have an explicit interface and be elemental.
I do not think this is needed because of 144:16-20.
260: 31-32
Why is this there, and how do we need to change this?

Type-bound procedures and procedure components:

55:10+ Add
(2+1/2) If the inherited binding is elemental then the overriding binding shall also be elemental.

Procedure dummy arguments:
263: 25 Remove C1229 and replace with
263: 30+
If procedure-name actual-arg is the name of a nonintrinsic elemental procedure, the dummy argument corresponding to actual-arg shall have an explicit interface and be elemental.
Is this enough and is there a better place to add this?
267: 25 Delete the parenthesized ending.

---

3. Add the dummy argument attribute SCALAR (and possibly NONSCALAR) that guarantee that the actual will be scalar.
This will also be considered part of the procedure characteristics. I do not give detailed edits here since this is subject to interpretation.

Either
4a. Allow INTENT(OUT) SCALAR arguments, or even better,
4b. Remove the purity requirement on elementals and replace with the condition that order of execution be unimportant to the user.
The random-number generation example above illustrates the fact that the requirements that the results of the execution of an elemental reference be completely independent of the order of execution is too strong. Consider a case where a user just wants an error flag raised if an error occurs during the execution of an elemental reference. The user does *not* want an array of flags (think exceptions). Solution 4a above provides an alternative.

Extended Example

Here is another example which attempts to illustrate the occurence of non-inlinable ELEMENTALs in codes which require the user to provide some procedure, which will be called on an array of non-zero rank, but the user does not need to know this. It is definitely not a desirable thing to ask the user to write 8 specific routines himself, as this is a hard and error-prone task.

The example is one of numerical quadrature (integration) of a user-provided function f(x) on the interval [a,b], i.e., the evaluation of a definite integral (DI). An OOP solution is given. Here f(x) needs to be evaluated on a whole grid of points (the integration mesh), which happens to be two-dimensional because it represents some kind of hierarchical fancy mesh.

The code is not documented in detail. Feel free to ask me about it.

First we design an abstract specification for an OOP definite integrator:

```
module DI_Quadrature ! DI=Definite Integral

   type, extensible :: di_problem
      ! The definite integral that is needed
      ! This will contain user-data upon extension
      real :: x_lb=0.0, x_ub=1.0 ! Bounds of integral
      ...
   contains
      procedure(f), pass(di), deferred :: f
         ! User provided f(x)
      ...
   end type di_problem

   type, extensible :: di_integrator
      ! The actual numerical integrator
      real :: global_error=0.0 ! Estimated error
      ...
```

```
   contains
      procedure(Integrate), pass(integrator), deferred :: Integrate
         ! The actual integrator--Euler, Runge-Kutta, etc.
      ...
   end type di_problem

   abstract interface
      elemental function f(problem,x)
         class(di_problem), intent(inout) :: problem
         real, intent(in) :: x
         real :: f
      end function f
      subroutine Integrate(problem,integrator)
         class(di_problem), intent(in) :: problem
         class(di_integrator), intent(inout) :: integrator
      end subroutine Integrate
   end interface

end module DI_Quadrature
```

Now we actually implement a specific fancy integration scheme:

```
module DI_Fancy_Quadrature
   ! Specific implementation: Fancy Fancy quadrature for DI's
   use DI_Quadrature

   type, extends(di_integrator) :: fancy_di_integrator
      integer :: n_points=0 ! Number of needed points along x axis
      real, dimension(:,:), allocatable :: x
         ! These are rank-2 for *this* integrator only
      ...
   contains
      procedure :: Integrate=>IntegrateFancy5
      ...
   end type fancy_di_integrator

contains

   subroutine IntegrateFancy5(problem,integrator)
      class(di_problem), intent(in) :: problem
      class(fancy_di_integrator), intent(inout) :: integrator

      real, dimension(5,integrator%n_points), allocatable :: f_x
         ! f(x) on a fancy grid
      ...
      f_x=problem%f(integrator%x) ! Call the elemental routine
         ! This will call the rank-2 specific ``elemental''
      ...
   end subroutine

end module DI_Fancy_Quadrature
```

The user on the other hand writes a description of his integration problem (in this case f(x)=a*x+x^b) for some user-specified a and b:

```
module My_DI
   use DI_Quadrature
   ! We do *not* use DI_Fancy_Quadrature since this may not yet be written

   type, extends(di_problem) :: my_di_problem
      real :: a=1.0, b=1.0 ! *Toy* example
      ! In real life there would be something *complicated* here
      ! such as a Monte-Carlo simulator to obtain f(x)
   contains
      procedure :: f=>my_f
```

```
   end type my_di_problem

contains

   elemental function my_f(di_problem,x)
      class(my_di_problem), intent(inout) :: my_di
      real, intent(in) :: x
      real :: f
      f=my_di%a*x+x**(my_di%b) ! f(x)=ax+x^b
   end function my_f

end module My_DI
```

And here is how this can actually be used in a program:

```
program DI_Example
   use DI_Quadrature
   use DI_Fancy_Quadrature

   type(my_di_problem) :: problem
   type(fancy_di_integrator) :: integrator

   problem%a=2.0; problem%b=-2.0
     // Make a specific DI
   ...
   call integrator%integrate(problem)
      ! You can also try another integrator if you have one
   ...

end program DI_Example
```