Subject: Coroutines (again)
From: Van Snyder
Reference: 03-258r1, section 1.1; 04-149r1

1 Number

TBD

2 Title

Coroutines.

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide for coroutines.

6 Rationale

In many cases when a "library" procedure needs access to user-provided code, the user-provided code needs access to entities of which the libary procedure is unaware. There are at least four ways by which the user-provided code can gain access to these entities:

- The user-provided code can be implemented as a procedure that is invoked either directly or by way of a dummy procedure, the extra entities can be made public entities of some module, and accessed in the user-provided procedure by use association.
- The user-provided code can be implemented as a procedure that is invoked either directly or by way of a dummy procedure, and the extra entities can be put into common if they're data objects.
- The user-provided code can be implemented as a procedure that takes a dummy argument of extensible type, which procedure is invoked either directly or by way of a dummy procedure, and the extra entities can be put into an extension of that type.
- The library procedure can provide for *reverse communication*, that is, when it needs access to user-provided code it returns instead of calling a procedure. When the user-provided code reinvokes the library procedure, it somehow finds its way back to the appropriate place.

Each of these solutions has drawbacks. Entities that are needlessly public increase maintenance expense. The maintenance expense of common is well known. If the user-provided procedure expects to find its extra information in an extension of the type of an argument passed through the library procedure, the dummy argument has to be polymorphic, and the user-provided code has to execute a SELECT TYPE construct to access the extension. Reverse communication causes a mess that requires GO TO statements to resume the library procedure where it left off, which in turn requires to simulate conventional control structures using GO TO statements. This reduces reliability and increases development and maintenance costs.

Reverse communication is, however, a blunt-force simulation of a well-behaved control structure that has been well-known to computer scientists for decades: The *coroutine*. Coroutines would allow user-provided code needed by library procedures more easily to gain access to entities of which the library

procedure is unaware, without causing the disruption of the control structure of the library procedure that reverse communication now causes.

I polled users of my library software for solutions of ordinary differential equations (both initial value and boundary value), evaluation of integrals by quadrature (in one and several dimensions), nonlinear least squares, nonlinear zero finding, and nonlinear optimization.

All of these packages provide for both forward ("call a subroutine") and reverse ("return when access to user code is needed") communication. The latter is a coroutine, heretofore implemented without syntactic support. The lack of syntactic support makes a mess of the control structure of these procedures.

I asked the users these questions:

1. Do you use forward or reverse communication?
2. On some arbitrary scale of your own devising, rate your model as "simple" or "complicated."
3. If you presently use reverse communication, and I revise my software to require you to use forward communication, and explain excellent new features in Fortran 2003 to support getting extra (non-state) parameters into your model, will it cause trouble for you?
4. If I were to revise my software so that in your reverse communication loop you would need to replace a CALL statement with a new RESUME statement, would it cause trouble for you?

Roughly half the users answered question #1 with "reverse." Of those, roughly 80% answered question #2 with "complicated." Almost all of the users who answered questions #1 and #2 in that way answered question #3 with, in essence, "Fu** off!" All of those who answered question #3 in that way answered question #4 with "not a problem."

Intrinsic support for coroutines would allow me to replace the internal control structures of my library routines that provide for reverse communication with ones that are far clearer and easier to understand, thereby reducing my long-term maintenance costs, without causing substantial cost for my users.

Coroutines are also useful to implement *iterators*, which are procedures that can be used both to enumerate the elements of a data structure and to control iteration of a loop that is processing those elements. Without coroutines, the way this is usually supported is to put the loop body into a subroutine, and pass that subroutine's name to the iterator. The problem with this is that it increases both development and maintenance costs. The subroutine that implements the loop body can't be an internal subroutine so one must either bundle up everything the loop references and put it into an extension of the type of some object the iterator passes through to the subroutine, or make everything more global than it deserves to be. Of course, these considerations apply equally to the user code needed by quadrature etc. software, but the case of a loop body makes the undesirability of packaging it as a separate subroutine more obvious.

7 Estimated Impact

Small. Minor additions to Section 12.

8 Detailed Specification

Provide two new statements, which we shall here call SUSPEND and RESUME,

If a subroutine suspends its execution by executing a SUSPEND statement, and its execution is subsequently resumed by executing a RESUME statement, execution resumes after the SUSPEND statement. Otherwise (either execution of the subroutine was terminated by execution of a RETURN or END statement, or it was invoked by a CALL statement), execution continues with the first executable statement of the invoked subroutine.

It is nonsense to allow a SUSPEND statement in a function, because there's no way to RESUME a function.

It would be reasonable to restrict coroutines to be nonrecursive, and to prohibit a SUSPEND and ENTRY statement to appear in the same subroutine.

A third statement, *viz.* COROUTINE could replace the SUBROUTINE statement, indicating that the program unit could contain a SUSPEND statement and could not contain an ENTRY statement. It may be necessary to add this statement in order for implementations to make dummy coroutines work. This could add some complication, as all references to the terms "subroutine" and "procedure" might need to be examined to determine whether it is necessary to add the term "coroutine" to the discussion. On the other hand, maybe it's enough to say "a coroutine is a subroutine that. . . ."

The RESUME statement need not appear in the same subprogram as the CALL statement that initiated execution of the coroutine.

It is not necessary or useful to prohibit internal subroutines to be coroutines.

Coroutines should be allowed to be type-bound procedures, actual arguments and procedure pointer targets. Generic coroutines should be allowed.

The question whether the entire instance of the procedure survives execution of a SUSPEND statement, or only those data entities that have the SAVE attribute survive, can be decided later. Similarly, the question whether modules and common blocks accessed from the coroutine survive can be decided later.

Fortran already has a limited form of coroutine: The relation between an input/output item list and a format is a coroutine relation. So it's not an entirely new concept for Fortran.

8.1 One possible implementation strategy

The main implementation problem is returning to the correct point of execution when a RESUME statement is executed. This can be accomplished by a hidden save variable to which the processor refers when control arrives in the procedure (recall that it is proposed to prohibit recursive coroutines). It is initialized by the processor to indicate the last exit was a result of a RETURN or END statement. If a SUSPEND statement is executed, the value of the hidden variable is changed to indicate that control should resume at the first executable statement after that SUSPEND statement. If control arrives in the procedure as a result of a CALL statement, or the hidden variable indicates that the last exit was a result of a RETURN or END statement, control proceeds to the first executable statement (recall that it is proposed that ENTRY statements not be allowed in coroutines). Otherwise, control proceeds to the point indicated by the hidden variable. The hidden variable's use could be similar to an assigned GO TO or to a computed GO TO, at the whim of the processor's developer.

A secondary problem is distinguishing whether control arrives in the procedure as a result of a CALL statement or a RESUME statement. This can be accomplished in at least two ways. One is to have a hidden argument that distinguishes the cases. The other is for the processor to generate two entry points, with appropriately mangled names, with one referenced by CALL statements and the other referenced by RESUME statements.

If the entire activation record survives a SUSPEND statement, it could be represented by a hidden saved pointer. It would be necessary to destroy a prior activation record if control arrives as a result of a CALL statement, to prevent memory leaks. The question whether the resurrected activation record includes information about argument association needs discussion. If so, then RESUME statements should not be permitted to have arguments. I prefer that the resurrected activation record not include information about argument association, because the absence of an argument list on the RESUME statement hides the dataflow from the human reader, and the semantics of VALUE become difficult to describe.

If the entire activation record does not survive a SUSPEND statement, there is still a question whether a SUSPEND statement should be allowed within a DO construct that has *loop-control* consisting of *do-variable* = *scalar-int-expr*, *scalar-int-expr*[, *scalar-int-expr*]. It is conceivable that processors could save and restore the hidden quantities inherent in the description in 8.1.6.4.1, but it may be easier simply to prohibit it.

No matter whether the entire activation record survives a SUSPEND statement or not, and if it does whether it includes argument association information, the specification part needs to be elaborated, at least to recreate automatic objects, because their extents and length parameters could depend on

common variables or variables accessed by use association (or host association in the case of internal coroutines).

8.2 Inferior alternative

An inferior alternative is to allow an ENTRY statement within a construct other than WHERE, FORALL or DO with *loop-control* consisting of *do-variable* = *scalar-int-expr*, *scalar-int-expr* [, *scalar-int-expr*]. This is inferior because it puts the onus on the user to return to the correct place in the library code. This could be ameliorated somewhat if the library routine has two layers — one that the user always calls by the same name, which looks at a flag that controls what's going on and re-invokes the "real" subroutine by the correct entry point. This slows things down. It is a step forward from the current situation because it doesn't require to disrupt the control structure to implement reverse communication. All in all, it's a relatively crappy solution.

9 History

This proposal was discussed and eventually rejected at meeting 166. The argument that led to its rejection was that one could always put the extra information for user-defined code into an extensible type. It was not considered at the time, however, that this requires the dummy argument of the user-provided subprogram to be polymorphic, and that the user-provided subprogram must execute a SELECT TYPE construct to gain access to the extra information. This overhead would not be necessary in a coroutine interaction. Furthermore, type extension cannot be applied to iterator construction. It was therefore brought up again at meeting 168, and rejected again, with the primary reason given being "we want to think about it some more." After gathering input from users, who urged resubmitting it, it was resubmitted at meeting 169.

10 Examples

In the simplest case, to simulate coroutines with existing facilities, one would have an argument that does dual duty to indicate what the caller is to do when the subroutine returns, and controls where the subroutine goes with it is called. Since the subroutine may return from several places for the same reason, the caller's decision-making process is messier than necessary. One usually instead has two arguments, a `what` argument that tells the caller what to do, and a `where` argument that keeps track of where the subroutine is to go when it gets control. The `where` argument is set by the caller to "start at the beginning," and if the caller changes it otherwise the guarantee is voided. The `where` argument can be eliminated by using two entry points in the subroutine: one to start it up, which sets a local saved `where` variable to "the beginning," and the other to continue processing.

The next two pages give examples of simulating coroutines with current facilities, and doing it directly using the proposed new facility. The simulation does not show whether `where` is an argument or a saved local variable, as outlined above. Assume that appropriate stuff is saved, either explicitly or because activation records are resurrected by RESUME statements.

10.1 Simulated

```
    go to ( 20, 60, 90 ), where
    ! Set up at beginning of problem, then
    what = function ! "what" is a dummy argument
10  continue
      if ( have enough function values for basic formula ) go to 30
      ! Get ready for a function value for basic quadrature step, then
      where = 1
      return
20    continue
      ! Add function * weight into quadrature estimate
    go to 10
30  if ( error estimate is small enough ) then
      where = 0
      what = good enough
      return
    end if
40  if ( another formula doesn't exist ) go to 70
50    continue
        ! if ( have enough function values for extended formula ) go to 30
        ! Get ready for a function value for extended quadrature step, then
        where = 2
        return
60      continue
        ! Add function * weight into quadrature estimate
      go to 50
70  continue
      ! Form a difference line of function values, then
80  continue
      if ( nothing goofy in difference line ) then
        ! Subdivide interval if it looks like smaller error will be
        ! achieved (control structure for this not shown), or
        what = done, but not as good as you asked for
        where = 0
        return
      end if
      if ( abscissa of difficult behavior sufficiently well isolated ) then
        ! subdivide the interval -- control structure for this not shown
      end if
      ! Decide where to add a point to difference line to search for
      ! difficult behavior, then
      where = 3
      what = function
      return
90    continue
      ! add function value to difference line
    go to 80

! Caller:
    where = 0
    do
      call quadrature ( a, b, answer, tol, err, what, where )
      if ( what /= function ) exit
      ! evaluate function
    end do
```

Aside from two IF ... THEN ... END IF blocks, this could be Fortran 66 code. The control flow is "hiding" in the value of `where`.

10.2 With coroutines

```
    ! Set up at beginning of problem, then
    what = function ! "what" is a dummy argument
    do while ( need a function value for basic formula )
      ! Get ready for a function value for basic quadrature step, then
      suspend
      ! Add function * weight into quadrature estimate
    end do
    do
      if ( error estimate is small enough ) then
        what = good enough
        return
      end if
      if ( another formula does not exist ) exit
      do while ( need a function value for extended formula )
        ! Get ready for a function value for extended quadrature step, then
        suspend
        ! Add function * weight into quadrature estimate
      end do
    end do
    ! Form a difference line of function values, then
    do
      if ( nothing goofy in difference line ) then
        ! Subdivide interval if it looks like smaller error will be
        ! achieved (control structure for this not shown), or
        what = done, but not as good as you asked for
        return
      end if
      if ( abscissa of difficult behavior sufficiently well isolated ) then
        ! subdivide the interval -- control structure for this not shown
      end if
      ! Decide where to add a point to difference line to search for
      ! difficult behavior, then
      what = function
      suspend
      ! add function value to difference line
    end do

! Caller:
    call quadrature ( a, b, answer, tol, err, what )
    do while ( what == function )
      ! evaluate function
      resume quadrature ( a, b, answer, tol, err, what )
    end do
```

The control flow is obvious. Which would you prefer to write and maintain?