Subject: Coroutines
From: Van Snyder
Reference: 03-258r1, section 1.1

1 Number

TBD

2 Title

Coroutines.

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide for corouties.

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. Ihere 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 compromises the ability to use well-structure control constructs.

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.

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.

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 subrutine 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 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. This would add some complication, as all references to the terms "subroutine" and "procedure" would need to be examined to determine whether it is necessary to add the term "coroutine" to the discussion. The RESUME atatement 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 actual arguments and procedure pointer targets.

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.

8.1 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. 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.