Communicating With the User in Numerical Routines

Part I: 00P Approach in F2x

Aleksandar Donev

Physics and Astronomy Department, Michigan State University
East Lansing, MI 48825
donev@pa.msu.edu

1 Preliminaries: Code in FWEB

Before I start walking through some Fortran code, let me point out to the reader that this document was written in the preprocessor/pretyprinter FWEB, available at,

http://w3.pppl.gov/~krommes

which enabled the fancy code formatting you are about to see. I had to make some workarounds since FWEB does not recognize F2x and F95 syntax fully, so do not think these are syntax errors. For example, ⇒ NULL() is replaced with \_\_NULL and TYPE is replaced with \_CASE\_TYPE in type selection blocks.

Code can be broken into segments (sections) and presented out of order, and later combined into proper Fortran code, so one needs to pay some attention to the code sections names, enclosed between @< and @>, and the section numbers. FWEB has very powerful macro capabilities as well, but I will not directly use these. Also note that comments are all C-style, following /// for short comments and enclosed in /* */ for long comments. I strongly encourage all programmers to take a look at FWEB!

2 Introduction: The Perils of Reverse Communication

Numerical algorithms very frequently require the user to provide certain operations and data which are rather generic in their specification, such as a function. For example, numerical quadrature involves the integration of a function \( f(x) \) over a specified range of values of \( x \). The value of such a function is likely to depend on other user data. The communication problem lies in finding a design that allows all users, with widely varying other data, to be accommodated. Other examples include solving equations and optimization.

The traditional solution to the communication problem, inherited from FORTRAN 77 libraries, is reverse communication. In this approach (which I shall not explicitly illustrate because it is so common), when communication with the user is required, such as say the value \( f(0.0) \), the numerical routine returns control
to the user along with a description of what is needed, such as the value 0.0 and a variable to place the result \( f(0.0) \) in, and an indicator saying that function evaluation is required. The user, in this case the program unit that called the numerical routine, performs this calculation and re-enters the numerical routine. Old codes using reverse-communication are among the most entangled spaghetti code, with numerous computed GOTO statements, alternate RETURN points, etc. With the flow-control constructs of Fortran 95 (called F95 from now on), reverse communication codes can be made nicer, but the same principle remains—return, compute, and re-enter.

I will not dwell longer on reasons why reverse communication is far from an ideal solution. I believe that reverse communication is a very sloppy solution to a very pervasive problem which has mysteriously remained unsolved in the last two revisions of Fortran. Almost all objections to reverse communication come from the fact that resources created inside the numerical routine need to be explicitly saved and restored or recreated each time the routine is re-entered. In F77 legacy codes, reverse-communication library routines are filled with SAVE attributes on local variables that need to retain their values between returns and re-entries. With modern dynamic allocation, where local variable space is allocated on the stack on a per-call basis, larger overheads are created, the compiler can do less optimizations because of the saved variables, etc.

This becomes particularly important and evident when considering parallelization. On an SMP machine for example, one may want to run multiple copies of the numerical routine concurrently, each running on a different thread. Such multiple copies would interfere with each other in the reverse communication model with SAVE, since such variables are necessarily statically allocated. Also, the routine may need to create (spawn) a pool of threads internally to do the calculations. Each time the routine returns, it is likely that the pool of threads will be released to the operating system, and will need to be recreated again upon the next entry. This is obviously a great overhead and makes the program volatile.

The problems with SAVE can be avoided by placing all working data into argument arrays that are passed to and from on every call. This is now thread safe, but data that should be private to the numerical routine is exposed to accidental corruption by the user and local scalar variables have to be copied to and from these arrays on every call (unless a large number of arguments is employed).

Reverse communication may have some benefits as well. Some people find it easy to use because all the user calculations are performed inside the user’s own programming environment, with no interfacing issues in the way. Also, it gives the user some more flow control, such as the ability to terminate the (usually iterative) numerical process without resorting to STOP and in general monitor the progress of the numerical routine easily. But I am convinced that reverse communication should be a solution of the past, and in this paper I will show how to solve the communication problem elegantly within the OOP framework.

3 The Proposed Solution: An OOP Approach

In this article I will illustrate with example code segments how Fortran 200x (called F2x from now on), as specified in the current draft of the proposal for the new standard, can be used to provide a truly robust, versatile and elegant communication mechanism, using polymorphic variables to emulate generic handles (generic pointers). I will call this handle-based communication, or for historical reasons, forward communication. A real application of this approach to an iterative linear solver, emulated in Fortran 95 (called F95 from now on), will be illustrated in a separate paper with fully-functional code.

3.1 An Example: Finding a Zero of a Polynomial
As an example I will write interfaces for a function \textit{FindPolyRoot} that finds one of the roots of a given polynomial
\[ p(x) = \sum_{k=0}^{n} a_k x^k. \]
I will represent a polynomial with a simple derived data type containing the coefficients of the polynomial:

\[
\langle \text{Polynomial} \, \text{3.1.1} \rangle \equiv \n\]

\begin{verbatim}
TYPE Polynomial
  REAL, DIMENSION (:), ALLOCATABLE :: coefficients // a_i--Requires ISO TR15581 extensions
ENDTYPE
\end{verbatim}

See also sections 3.3.1 and 3.3.7.

I will assume that I have at my disposal general-purpose non-linear solver routine \textit{FindRoot} that can find the root of a function \( f(x) \) in a given interval \([a, b]\), and that \textit{FindRoot} only requires evaluation of \( f \) at certain points \( x \) generated during the search for the root. \textit{FindPolyRoot} will thus internally calculate a suitable root bracketing interval \([a, b]\) (I won’t give any numerical codes here, just interfaces) and call \textit{FindRoot} to actually compute the root. This is a typical situation that arises often and that is sufficient to fully illustrate the communication problem. My (oversimplified) interface for \textit{FindPolyRoot} will be:

\[
\langle \text{FindPolyRoot} \, \text{3.1.2} \rangle \equiv \n\]

\begin{verbatim}
INTERFACE
  FUNCTION FindPolyRoot (polynom) RESULT(zero)
    TYPE (Polynomial), INTENT(IN) :: polynom // p(x)
    REAL :: zero // Zero x such that p(x) = 0
  END FUNCTION FindPolyRoot
END INTERFACE
\end{verbatim}

The problem now is how to interface the procedure \textit{FindRoot} so that it can find the root of any function, including polynomials, and how to call it from within \textit{FindPolyRoot}. It should be evident that object-oriented programming features are relevant to this example. Function of a single variable \( f(x) \) form a class of objects, with polynomials \( p(x) \) being members of this class. I will not discuss OOP features or F2x here since this has been done at length in previous editions of the \textit{ACM Fortran Forum}. A wonderful reference is the article “Object Orientation and Fortran 2002: Part II” by Malcolm Cohen in volume 18. But before I give an example of an OOP-based communication mechanism, let me point to a F95 (and F2x) alternative based on module (or host) association.

### 3.2 Working Within F95: Association-Based Communication

In order to have a certain uniformity in the code segments, I will agree that the function \( f(x) \) should be provided to \textit{FindRoot} in the form of a dummy procedure argument \textit{EvaluateFunction}, which among other
things will accept the value of $x$ and return the value $f(x)$. The simplest possible interface of \texttt{FindRoot} will thus be (along with some traditional arguments of root-finding routines):

\[
\langle \texttt{FindRoot 3.2.1} \rangle \equiv
\]

\begin{verbatim}
  INTERFACE SUBROUTINE FindRoot(EvaluateFunction, root, interval, tolerance, success)
    INTERFACE
      REAL FUNCTION EvaluateFunction(x)
      REAL, INTENT(IN) :: x
    END FUNCTION EvaluateFunction
  END INTERFACE
  REAL, INTENT(INOUT) :: root   // Initial guess for $x$ on input, root of $f(x) = 0$ on output
  REAL, INTENT(IN), DIMENSION(2) :: interval   // $[a, b]$  
  REAL, INTENT(IN) :: tolerance  // Desired precision
  LOGICAL, INTENT(OUT) :: success  // $\mathcal{F}$ if solution converged, $\mathcal{F}$ otherwise
  END SUBROUTINE FindRoot
END INTERFACE
\end{verbatim}

See also sections 3.3.3 and 3.3.9.

\texttt{FindPolyRoot} will pass on to \texttt{FindRoot} as an actual argument for the dummy \texttt{EvaluateFunction} a function called \texttt{EvaluatePolynomial}. A problem arises: \texttt{EvaluatePolynomial} will need to evaluate $p(x)$ for a given $x$ and thus require access to the argument \texttt{polyom} of \texttt{FindPolyRoot}, which is not passed to it in its own argument list. A possible solution, which will work fine in most situations, is to make a global variable or pointer (say a module variable) of type \texttt{TYPE(Polynomial)}, that both \texttt{FindPolyRoot} and \texttt{EvaluatePolynomial} will share, in which \texttt{FindPolyRoot} will store the value of \texttt{polyom} before calling \texttt{FindRoot}. I can not even list all the problems with this approach. Other than the unnatural “behind-the-scenes” sharing of the argument \texttt{polyom} between two procedures, and the evident lack of elegance in the approach, it should be clear that only one instance of the function \texttt{EvaluatePolynomial} and \texttt{FindPolyRoot} can be active at a given moment because of the access to global (static) data. This is a very unnatural limitation that becomes important in the context of concurrent (parallel) computing.

If one is given the possibility of passing internal procedures as actual arguments to routines that have dummy procedure arguments, then \texttt{EvaluatePolynomial} could be an internal procedure of \texttt{FindPolyRoot} and thus have access to \texttt{polyom} via host association. With this possibility, reverse communication can be elegantly avoided in almost all practical cases of interest, while still keeping all of the advantages of reverse communication, such as the familiar programming environment of the host for coding \texttt{EvaluateFunction}. The internal procedure can do pretty much all the things that a full return to the calling routine could do, but with the added benefits of an effective and flexible communication interface. At present, internal procedures are forbidden from being actual arguments in F2x, and passing an internal procedure as an actual argument can not be emulated in F95, so I will not discuss this any further. As someone who cares about this issue greatly though, I would urge the J3 committee to look at this issue closely. It truly deserves all the attention it can get!

Another very important problem with the above interface for \texttt{FindRoot} is the fact that the interface of \texttt{EvaluatePolynomial} was pre-determined and fixed, so that this routine had to be written specifically for use with \texttt{FindPolyRoot}. Of course, we could have written another less-specialized procedure for evaluating a polynomial, which would accept a polynomial as an argument and a value for $x$, and then have
EvaluatePolynomial invoke (wrap) this procedure. This would add yet another clumsy layer of interfacing. In summary, by fixing the interface for \( f(x) \), the options of the user in calculating \( f(x) \) are much restricted, and there are situations where this is too restrictive. In a sense, reverse communication gives too much freedom to the user by completely returning control out of the numerical routine, while a fixed interface for a user-supplied function is too restrictive. We need to find a balance between flexibility and sufficiency, and of course always bear in mind efficiency. My solution basically uses polymorphic arguments to make the interface for \( f(x) \) flexible enough for most purposes, yet compact enough not to require a return to the calling program unit. I illustrate this next and conclude this paper.

### 3.3 Looking Toward F2x: Handle-Based Communication

The above approach of supplying \( f(x) \) as a dummy procedure argument to FindRoot is in my opinion rather satisfactory and translates well into F95. The only problem was the fact the we needed to pass on a polynomial to EvaluatePolynomial, but when coding FindRoot we could not have predicted that the actual function whose root we are finding the root would be a polynomial, so we could not write the interface of EvaluateFunction accordingly. The solution is conceptually simple: Let FindPolyRoot pass on to FindRoot a handle for the function \( f(x) \) that needs to be evaluated, and then this handle can be passed on to EvaluateFunction, so that the actual function-evaluation procedure EvaluatePolynomial can deduce which polynomial it is evaluating. In a sense, one can think of a handle as an object that uniquely defines a member (such as a specific function \( f(x) \)) of a given set of objects (such as all one-argument functions \( f(x) \) under consideration). The handles will be agreed upon between FindPolyRoot and EvaluatePolynomial, but FindPolyRoot need not know what this handle means!

In F2x, the handle can be a polymorphic dummy argument (or polymorphic pointer) to an object of the extensible data type One_Argument_Function, and the actual argument (or target of the pointer) will be of a type Polynomial that extends One_Argument_Function. I prefer to think of this as a specific implementation of the generic handle idea because handles can be emulated in non-OOP languages such as F95, at the cost of elegance and some functionality. Example code will illustrate this handle-based communication best. Please note that I had to change the syntax of extensible-type declaration into illegal F2x to keep proper formatting by FWEB:

\[
\text{(Polynomial 3.1.1) +≡}
\]

```
TYPE One_Argument_Function    // A generic \( f(x) \)
EXTENSIBLE        // This should go on the line above
    REAL, DIMENSION (2) :: range    // The range of \( f \) in \( \mathbb{R} \)—just an example component
ENDTYPE

TYPE Polynomial    // A polynomial \( p(x) \)
    EXTENDS (One_Argument_Function)     // Move to line above
    REAL, DIMENSION (:), ALLOCATABLE :: coefficients    // The coefficients \( a_i \)
ENDTYPE
```

Let me only briefly explain the base F2x OOP features used in this paper to those unfamiliar with F2x. A derived data type with the \texttt{EXTENSIBLE} attribute forms the base of a given class and can be extended with
some additional data later on, and any data type that extends it will belong to the class of the base type and inherit all of the base components. A class object does not have a fixed type as do all F95 variables (strong typing), so it is a polymorphic type. Pointers or dummy procedure arguments can be polymorphic, and they can be associated with any target or actual argument that is of the correct class, at run-time.

Components of polymorphic types can also be procedures, so called type-bound procedures, declared after a contains in the derived type definition, or procedure pointers, which are also new to F2x. Interfaces for procedures can be made with interface procedure, which fixes an error-prone annoyance in F95 where one has to repeat an interface rather than just import it from an interface body via host association, and also provides argument checks for procedures invoked via pointers as well. Derived type definitions can also be imported via import in F2x inside interfaces, to avoid F95 redundant repeating. Type selection with select type can be used to branch according to the actual run-time type of a polymorphic variable. Please see previous articles in the Fortran Forum for a more complete description.

Using polymorphic variables as generic handles, a (shortened) possible interface for FindRoot would be:

\[ \langle \text{FindRoot 3.2.1} \rangle \equiv \]

\[
\begin{align*}
\text{INTERFACE} &\quad // I\text{ ommitted some of the arguments for brevity and simplicity} \\
\text{SUBROUTINE} &\quad \text{FindRoot} (\text{function}, \text{root}, \text{EvaluateFunction}) \\
\text{CLASS} &\quad (\text{One} \_ \text{Argument} \_ \text{Function}), \text{INTENT} (\text{IN}) :: \text{function} \_ \quad // \text{The function to evaluate} \\
\text{REAL, INTENT} &\quad (\text{INOUT}) :: \text{root} \quad // \text{Initial guess for x on input, root of f(x) = 0 on output} \\
\text{INTERFACE} &\quad // \text{Observe the added argument function} \_ \\
\text{REAL FUNCTION} &\quad \text{EvaluateFunction} (\text{function}, x) \quad // \text{Function evaluator} \\
\text{CLASS} &\quad (\text{One} \_ \text{Argument} \_ \text{Function}), \text{INTENT} (\text{IN}) :: \text{function} \_ \\
\text{REAL, INTENT} &\quad (\text{IN}) :: x \\
\text{END FUNCTION} &\quad \text{EvaluateFunction} \\
\text{END INTERFACE} \\
\text{END SUBROUTINE} &\quad \text{FindRoot} \\
\text{END INTERFACE}
\end{align*}
\]

I must emphasize that this is only one of several possible organizations of the data types and interfaces. One of its advantages, which is a natural approach in some cases (such as the iterative linear solver illustrated in the sequel to this paper), is that the same function evaluation routine can be used for several different types of functions, probably using type selection in F2x. The above example illustrates all my points and gives the general direction in which I think communication interfaces should be taken in the numerical community. Also, the above approach can at least partially be emulated in F95, as I will illustrate with a fully functional F95 code for an iterative linear solver in a separate paper.

In the specific case of root finding however, evaluation is a fundamental property of a function, so it should somehow be included as a part of the data type One_Ar gument Function. The code below makes the evaluation procedure type-bound, which is just a syntactically higher way of adding a component to One_A r gument Function that is a pointer to a function with an interface similar to that of EvaluateFunction in the above code segment. How exactly the organization of One_A r gument Function is done is largely a matter of choice, but the basic principle will remain—the evaluation procedure will accept a polymorphic argument of type One_A r gument Function. Unfortunately the approach below can not be emulated in F95 (because procedure pointers do not exist in F95).
\{FunctionHandles 3.3.5\} ≡

\textbf{TYPE} \ One\_Argument\_Function \ \ // \ A \ generic \ f(x)
\textbf{EXTENSIBLE} \ \ // \ Move \ above
\textbf{REAL, DIMENSION (2)} :: \ \textit{range} \ \ // \ The \ range \ of \ f
\textbf{CONTAINS}
\textbf{PROCEDURE} (\textit{EvaluateFunction}), \ \textbf{PASS} \ \textit{OBJ} :: \ \textit{Evaluate} = \ \_\textit{NULL}
\ \ // \ A \ dummy \ evaluation \ procedure
\textbf{ENDTYPE}

\textbf{INTERFACE \ PROCEDURE} (\ )
\textbf{REAL FUNCTION} \ \textit{EvaluateFunction}(function, x) \ \ // \ The \ interface \ of \ a \ function \ evaluator
\textbf{CLASS} (\One\_Argument\_Function), \ \textbf{INTENT} (IN) :: \ \textit{function} \ // \ Implicitly \ passed
\textbf{REAL, INTENT} (IN) :: \ \textit{x} \ \ // \ The \ argument \ \textit{x}
\textbf{END FUNCTION} \ \textit{EvaluateFunction}
\textbf{END \ INTERFACE}

Now we can extend \One\_Argument\_Function to a data type for a polynomial and override the evaluation procedure with a specific procedure for evaluating polynomials (such as one using Horner’s Rule), called \_\textit{EvaluatePolynomial} in the code below:

\{Polynomial 3.1.1\} +≡

\textbf{TYPE} \ \textit{Polynomial} \ \ // \ A \ polynomial \ \textit{p(x)}
\textbf{EXTENDS} (\One\_Argument\_Function) \ \ // \ Move \ above
\textbf{REAL, DIMENSION (:)}, \ \textbf{ALLOCATABLE} :: \ \textit{coefficients} \ \ // \ The \ coefficients \ \textit{a_i}
\textbf{CONTAINS}
\ \ // \ Override \ the \ evaluation \ procedure
\textbf{PROCEDURE} (\textit{EvaluatePolynomial}), \ \textbf{PASS} \ \textit{OBJ} :: \ \textit{Evaluate} = \ \_\textit{EvaluatePolynomial}
\ \ // \ = \ should \ be \ \Rightarrow
\textbf{ENDTYPE}

\textbf{INTERFACE \ PROCEDURE} (\ )
\textbf{REAL FUNCTION} \ \textit{EvaluatePolynomial}(\textit{polyynom}, x) \ \ // \ A \ polynomial \ evaluator
\textbf{CLASS} (\textit{Polynomial}), \ \textbf{INTENT} (IN) :: \ \textit{polyynom} \ \ // \ The \ polynomial
\textbf{REAL, INTENT} (IN) :: \ \textit{x} \ \ // \ The \ argument
\textbf{END FUNCTION} \ \textit{EvaluatePolynomial}
\textbf{END \ INTERFACE}

And here is the new interface to \textit{FindRoot}. Inside \textit{FindRoot}, when function evaluation is needed, \textit{EvaluateFunction} will be substituted with \textit{function\_\%\textunderscore Evaluate}, and the actual argument that \textit{FindPolyRoot} will pass to \textit{FindRoot} for the dummy \textit{function\_\%} will be of actual type \textit{Polynomial}:
\langle \text{FindRoot 3.2.1} \rangle + \equiv

\text{INTERFACE}

\text{SUBROUTINE } \text{FindRoot (function\_ root)}
\begin{align*}
\text{CLASS (One\_Argument\_Function), INTENT (IN) :: function\_} & \quad \text{// The function to evaluate} \\
\text{REAL, INTENT (INOUT) :: root} & \quad \text{// Initial guess for } x \text{ on input, root of } f(x) = 0 \text{ on output} \\
& \quad \text{// Other arguments will be unchanged...}
\end{align*}
\text{END SUBROUTINE } \text{FindRoot}
\text{END INTERFACE}

4 Conclusions

I will finish this paper’s Conclusions, References and Acknowledgements in its related sequel. I hope some discussion of the communication problem will follow on the comp-fortran-90 list, and I welcome any critique and suggestions for improvement.