Ftcl: combining Fortran and Tcl

Introduction

Fortran seems to be one of those programming languages that everybody knows nobody should use. At least: it does not receive much (positive) attention from computer scientists, it hardly ever makes it to the popular computer magazines and it has a reputation of being an old and therefore useless language. Reliable numbers on actual use are hard to come by - for any programming language, but here are a few facts that oppose the common view:

- The current standard, Fortran 95, is a much different language than FORTRAN 77, the version that everyone "knows": it has many features that exploit modern ideas in programming and of current computers (see the appendix for a short overview).
- A new major revision is on it s way, Fortran 2003, so the language keeps evolving.
- If engineers do not program in Fortran they are more likely to use such systems as MATLAB or Scilab than languages like C, C++ or Java.

Fortran continues to be a very useful language when it comes to number crunching, but it falls short when it comes to tasks such as (graphical) user-interfaces, plotting facilities and network capabilities. One reason for this might be that there are no platform-independent and vendor-independent libraries. Every compiler comes with its own package, if any, and numerous ad hoc or partial solutions can be found.

The subject of this paper, the "Ftcl" library, is one such solution: it uses Tcl to provide the missing facilities to Fortran programs and alternatively it provides number-crunching capabilities to Tcl programs via Fortran. The paper describes the basic idea behind the interfacing, the design choices, two small examples and it discusses the work that is currently being done.

Design of Ftcl

The requirements for the "Ftcl" library are simple: a typical Fortran programmer who wants to use Tcl does not want to become involved in the C API - the less C programming the better. And this is exactly what you can achieve with it: you can program in Fortran and in Tcl, almost entirely without the need for C. The library has not been set up to make all Tcl functions available in Fortran though. The emphasis is on data exchange:

- Routines of the ftcl_get_* and ftcl_put_* family allow you to get and set the value of Tcl variables.
- A routine ftcl_script runs a Tcl script after which you can query the results as stored in Tcl variables.

Besides these relatively simple routines, it is possible to use the library for two different programming models:

- You can regard the library as a tool inside a Fortran program, that is the application is driven by a Fortran main program that allows scripting facilities. Examples are:
    - Running a computational program that sends its results over sockets to a visualisation program on another machine.
    - Replacing a textual interface by a graphical one, this may require some restructuring to allow event processing, but that is fairly easy to achieve in many cases.
- You can also use the library to create an extension to Tcl written in Fortran, so that computations of all sorts can be driven by means of Tcl's scripting facilities. One might think of well-known libraries such as BLAS or LAPACK, but also of legacy libraries or of completely replacing the Fortran main program by a script.

Exchanging data

When interfacing between two programming languages you are always facing problems such as calling conventions and the proper translation of data types. Luckily, for the interfacing between C and Fortran the issues are well-known and have been solved satisfactorily, albeit in a compiler-dependent way:

- calling conventions may require you to be careful with the "name" of the function or subroutine and with any hidden arguments[1] (*). This can usually be solved with appropriate C macros.
- data types such as floating-point numbers and integers do not present a problem - except that matching types must be chosen (again the burden is on the C side)
- data types such as character strings and logicals require some conversion: Fortran does not use a trailing NUL character to indicate the end of the string, but enforces a maximum length and pads strings with spaces.
- arrays should be passed from C to Fortran and vice versa via the old FORTRAN 77 style: this simply passes the starting address and does not pass additional information, such as the dimensions.

1 Fortran strings are usually passed with a hidden length argument.

- if at all possible, you should not pass pointers or structures: C's concept of a pointer is unrelated to Fortran pointers - these are more like object references in Java or C++. And C structs and Fortran derived types are not guaranteed to have the same layout.

With these restrictions in mind, the Ftcl library has a set of routines that transfer the data from a Tcl variable to a Fortran variable and vice versa. When using the generic interfaces, this may look like:

```
   use Ftcl      ! Use the Ftcl module, so that interfaces are
                 ! properly checked
   real    :: value
   integer :: count

   ...
   call ftcl_get( 'count', count )
   call ftcl_get( 'factor', value )
```

where the specific routine (ftcl_get_int or ftcl_get_real) is chosen by the compiler based on the type of the Fortran variable. This also causes the value held by the Tcl variable to be converted into a real or an integer. In this way, the difference between Tcl's dynamic and Fortran's static typing is solved. (Currently, errors are handled silently - see also the last section).

For scalar data this approach works well, as there is little to chose. For arrays of data, arrays in the Fortran or C sense, it is a different matter: You can argue that Fortran arrays should be matched with Tcl arrays on the ground of syntax, but with Tcl lists on the grounds of organisation. Or with byte arrays because they are more efficient in memory use and when exchanging large amounts of data between programs. As there is no single best approach, all three methods are, or will be, provided by the Ftcl library.

Multi-dimensional Fortran arrays pose an additional design choice:

```
   real, dimension(10,20) :: array
```

becomes:
- a Tcl array with elements "1,1", "2,1", ..., "10,1", ... "10,20"
- a Tcl list of 20 elements, each one a list of 10 floating-point numbers.
- a byte array that is organised as 20 pieces of 40 bytes (assuming a single-precision floating-point number occupies 4 bytes)

These choices reflect the organisation of the data on the Fortran side: the leftmost index runs fastest, in contrast to C.

When transferring such data from Tcl to Fortran, the Fortran variable is used to determine the dimensions. This way no memory overflow will occur.

Creating Tcl commands

Just transferring data may not always be enough. The Ftcl library also provides a facility to create new Tcl commands that are implemented via Fortran routines. This is achieved by the following procedure.

In the Fortran code you register a routine, much as you would with Tcl's C API:

```
module my_commands
   use ftcl
contains
subroutine square( cmdname, noargs, ierror )
   character(len=*)  :: cmdname
   integer           :: noargs
   integer           :: ierror

   ...

end subroutine
end module

program myprog

   use ftcl
   use my_commands

   ...

   call ftcl_make_command( square, 'calc_square' )

   ...
```

```
end program
```

The C code for the function "ftcl_make_command" is:[2]

```
FOR_RETURN FOR_CALL ftcl_make_command(
   FortProc       procedure,        /* I (pointer to) the Fortran procedure */
   char          *cmdname,          /* I name of the associated Tcl command */
   FTNLEN         length            /* I (implicit length of the string */
                          )         /* Returns nothing */
{
   char *pstr ;

   pstr = ftcl_conv_str_from_fort( cmdname, length ) ;
   Tcl_CreateObjCommand( ftcl_interp, pstr, Ftcl_GenericCmd,
      (ClientData) procedure, NULL ) ;

   RETURN ;
}
```

That is, the Fortran routine is registered as client data for a generic C function (Ftcl_GenericCmd). This function takes care of some administrative tasks and then calls the actual Fortran routine:

```
static int Ftcl_GenericCmd( ClientData client_data, Tcl_Interp *interp,
                            int objc, struct Tcl_Obj * CONST objv[] )
{
   integer  ierror                ; /* Passed to Fortran */
   int      length                ;
   char     cmdname[FTCL_BUFSIZE] ;
   char    *pstr                  ;
   FortProc procedure             ;

   ... Administrative tasks left out ...

/* Get the first argument and call the Fortran routine
*/
   pstr = Tcl_GetStringFromObj( objv[0], &length ) ;
   ftcl_conv_str_to_fort( pstr, cmdname, sizeof(cmdname) );

   procedure = (FortProc) client_data ;
   (*procedure)( cmdname,
#ifdef IN_BETWEEN
      (FTNLEN) sizeof( cmdname ),
#endif
      &ftcl_number_args,
      &ierror
#ifndef IN_BETWEEN
     ,(FTNLEN) sizeof( cmdname )
#endif
                                  ) ;
   if ( ierror != 0 )
   {
      Tcl_SetResult( interp, "error in ", NULL ) ;
      Tcl_AppendResult( interp, pstr, NULL ) ;
      return TCL_ERROR ;
   }
   else
   {
      return TCL_OK ;
   }
}
```

Because an array of C pointers can not be reliably used in Fortran, the typical argument "objv" for C functions that implement commands can not be passed directly. Instead you can get the value of the n-th argument via:

```
   integer :: myvar
   ...
   call ftcl_get_arg( n, myvar )
```

Depending on the type of the Fortran variable "myvar", an integer, real, string or whatever is returned.

Results can be returned via:

[2] The C fragments shown in this paper contain a number of macros and special data types that are used to make the interfacing to Fortran as easy and platform-independent as possible. It does not mean the code is easier to understand though.

```
   real :: computation_result
   ...
   call ftcl_set_result( computation_result )
```

The signature of a Fortran routine to be used as a Tcl command is:[3]

```
interface
      subroutine square( cmdname, noargs, ierror )
      character(len=*)  :: cmdname
      integer           :: noargs
      integer           :: ierror
   end subroutine
end interface
```

The first argument, cmdname, is the name of the Tcl command (useful for error reporting and such), the second is the number of arguments that were passed and the third is used to set the error code, as expected from any Tcl command.

Creating loadable extensions

A further goal of Ftcl is to create loadable extensions written in Fortran. This is also possible via the experimental Fortran version of Critcl [ref. S.Landers]), but Critclf relies on the presence of both a C and a Fortran compiler on the system that should run it.

In essence such an extension is not different from a C extension:

- you need an appropriate initialisation function that defines the specific Tcl commands.
- you need to create a DLL or a shared library.

Because of the "name" for the initialisation function it is inevitable at this point to do some C programming. Well, it is limited to filling in the name of an equivalent Fortran start-up routine in a small set of C macros:

```
#define C_INIT_NAME               Mypkg_Init
#define FORTRAN_PACKAGE           "Mypkg"
#define FORTRAN_START             mypkg_start
#define FORTRAN_START_ALLCAPS     MYPKG_start
#define FORTRAN_START_UNDERCORE   mypkg_start_
#define FORTRAN_START_DBL_UNDER   mypkg_start__

#define VERSION "1.0"
```

The Fortran program must replace the strings Mypkg, MYPKG and mypkg by the name of the package in the same combination of uppercase and lowercase letters: there are a lot of naming conventions for Fortran and C compilers. Via a set of compiler-defined macros, the correct name is chosen and the result put in the macro FORTRAN_START_NAME.

These macros are then used in a generic initialisation function:

```
#include "ftcl.h"
#include "ftempl.h"

EXPORT_FUNC int C_INIT_NAME( Tcl_Interp *interp ) ;
FOR_RETURN FOR_CALL FORTRAN_START_NAME( void ) ;

int C_INIT_NAME( Tcl_Interp *interp )
{
   int retcode ;
   int error   ;

/* Register the Fortran logical values
*/
   ftcl_init_log( &ftcl_true, &ftcl_false ) ;

/* Initialise the stubs
*/
#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8.0", 0) == NULL) {
       return TCL_ERROR;
    }
#endif
```

[3]Interface blocks have the same purpose in Fortran as prototypes in C. For subroutines (and functions) contained in modules the interface is defined automatically.

```
/* Inquire about the package's version
*/
    if (Tcl_PkgRequire(interp, "Tcl", TCL_VERSION, 0) == NULL) {
       if (TCL_VERSION[0] == '7') {
          if (Tcl_PkgRequire(interp, "Tcl", "8.0", 0) == NULL) {
             return TCL_ERROR;
          }
       }
    }

    if (Tcl_PkgProvide(interp, FORTRAN_PACKAGE, VERSION) != TCL_OK) {
       return TCL_ERROR;
    }

/* Register the package's commands
*/
   retcode = TCL_OK ;

   ftcl_interp = interp ;

   FORTRAN_START_NAME( &error ) ;
   if ( error != 0 ) {
      retcode = TCL_ERROR ;
   }

   return retcode ;
}
```

All that remains is the compiling and linking process. Ideally, Ftcl would use TEA3 for this, but the support for Fortran 95 within "autoconf" is very poor at this moment. Instead, Ftcl comes with examples of makefiles for various compilers to show how it should be done.

A completely different problem occurs when the programmer does not have access to a C compiler. In that case it is possible to use a "pre-cooked" extension called "Ftcl". This expects an initialisation routine with a fixed name, "ftcl_initialise". As the C part can be completely prepared for a given platform, there is no need for a C compiler after that.

Two small examples

I will explain the practical use of the library with two small examples: one that exploits sockets to transfer information to an external program and one that implements a user-interface for a computation.

The code below shows a straightforward (and actually trivial) computation:

```
program calc
   use FTCL
   implicit none

   integer       :: i
   integer       :: nosteps
   real          :: x
   real          :: y
   real          :: dphi
   real          :: phi

   call ftcl_start( 'calc.tcl' )
   call ftcl_get( 'dphi', dphi )
   call ftcl_get( 'nosteps', nosteps )

   write(*,*) dphi, nosteps

   do i = 0,nosteps
      phi = real(i) * dphi
      x = cos( phi )
      y = sin( phi )
      write(*,*) phi, x, y
      call ftcl_put( 'x', x )
      call ftcl_put( 'y', y )
      call ftcl_script( 'transfer' )
   enddo

   call ftcl_script( 'close_transfer' )
   stop
end program
```

It first initialises the Ftcl library (ftcl_start) and sources a script in the process. This script defines the values of two parameters for the computation. It also defines a procedure "transfer".

In a loop the Fortran program computes the x and y coordinates and calls the Tcl procedure "transfer". It is not at all important - to the Fortran program - what this command actually does: that is all handled independently of the program.

In this case, the transfer procedure and two auxiliary procedures look like this:

```
proc transfer { } {
   global channel
   global x
   global y
   puts $channel "$x [expr $y]"
   flush $channel
}
proc close_transfer { } {
   global channel
   close $channel
}

proc SetUp  { {host localhost} } {
   global channel
   set port    8085
   set channel [ socket $host $port ]
}

SetUp
set nosteps 100
set dphi    0.1
```

So:

- at start-up, the Tcl script sets up a connection to a little server program on the local host.
- the transfer procedure sends the current values of x and y to that server.
- the close_transfer procedure closes the connection.

This example illustrates how easy it is and how few lines of code it requires to do socket programming in Fortran - well, if you have Tcl at your disposal.

<<picture_lander.gif>>
Figure 1. The user-interface and display of the Lander example.

The second example, created by Clif Flynt to get a feel for the library and to see if it fit his needs, is the simulation of a lunar (terrestrial) lander. The computation remains fairly simple, but the complication now is that there is user input required via the GUI and that the result is visualised (see Figure 1):

```
      PROGRAM LANDER

      USE FTCL

      REAL :: IMPULSE
      CHARACTER(80) astring

      IMPULSE = 200
      FHEIGHT = 10000.0
      SPEED = 100.0
      FUEL = 1000.0
      GROSS = 900.0

      !
      ! Initialise the Ftcl library and load the Tk extension
      !
      CALL ftcl_start('config.tcl')
      CALL ftcl_script('package require Tk')
      CALL ftcl_script('inputInitialValues')
      CALL ftcl_script('update')
      CALL ftcl_script('update idle')

      CALL ftcl_get_string('burn', astring)

      !
      ! Wait until the user presses the Go button
      ! the Tcl variable "ready" will have a non-zero
      ! value then
      !
      CALL ftcl_get_int('ready', irdy)
      DO WHILE (irdy == 0)
        CALL ftcl_script('update')
        CALL ftcl_script('update idle')
        CALL ftcl_get_int('ready', irdy)
      ENDDO

      !
      ! Start the computation and the visualisation
      !
      CALL ftcl_get_real('burn', burn)
      I = 0

      DO WHILE (FHEIGHT .GT. 0)
        CALL ftcl_get_real('burn', burn)
        CALL CALCSPEED (SPEED, FUEL, GROSS, BURN, IMPULSE, SPEED)
        I = I + 1
        FUEL = FUEL - BURN
        IF (FUEL .LE. 0)  THEN
          FUEL = 0
          BURN = 0
          CALL ftcl_put_real('burn', burn)
        ENDIF
        FHEIGHT = FHEIGHT - SPEED
        CALL ftcl_put_int('time', I)
        CALL ftcl_put_real('speed', speed)
        CALL ftcl_put_real('fuel', fuel)
        CALL ftcl_put_real('ht', FHEIGHT)
        CALL ftcl_script('showState')
        CALL ftcl_script('update idle')
      ENDDO

      !
      ! Just let the user end the program
      !
      DO WHILE (1 .EQ. 1)
        CALL ftcl_script('update idle')
        CALL ftcl_script('update')
      ENDDO

      END

      SUBROUTINE CALCSPEED (finit, fuel, gross, burn, impulse, speed)
      REAL :: MASS, IMPULSE
      MASS = gross + fuel
!
!      speed += 9.8 * (1 - impulse * log(mass/(mass-burn)))
!
      SPEED = SPEED + 9.8 * (1 - impulse * log(mass/(mass-burn)))
      END
```

To make sure the GUI is "alive", you need to allow Tcl's event loop to run regularly, hence the calls to "update".

This is one strategy: keep the structure of the original program intact and make calls to Tcl/Tk at appropriate moments. A different approach would be to split the computation into small pieces and to let a script handle the overall loop (in this case: the subroutine CALCSPEED would become a Tcl command and the entire main program would be replaced by a script). The various options choices have been described before with the combination C-Tcl [ref. Flynt].

Deployment issues

In its current state the Ftcl library poses a number of issues when running programs that use it:

- At start-up the shared or dynamic libraries for Tcl must be found. Of course this is an issue with most if not all applications nowadays. This may include the Fortran runtime libraries.
- The program must also find the initial script files. One solution is to keep them in a "lib" subdirectory of the directory in which the program is started or installed, but a much better solution is to store the initial script files (and possibly the application's script too) in the executable itself - a technique such as "TOBE" (http://wiki.tcl.tk/7908) can be used to make the executable "stand-alone".

Conclusion

The library I have described in this paper is a very useful tool for combining Fortran and Tcl into one system: the two languages are complementary to each other, as they emphasize differnet aspects of programming.

The library is not mature yet in the sense that deployment is as smooth as it can be with an ordinary Tcl application, nor is its functionality complete. For instance support for creating loadable extensions is in its infancy and catching erors is currently a bit cumbersome (you would have to examine the errorCode and errorInfo variables).

Work on this library continues (and at this point I would like to thank Clif Flynt for his valuable contributions), but it has proven its usefulness already.

References

Clif Flynt (2003)
Embedding the Tcl interpreter for portable application development
4th European Tcl/Tk User Meeting, Nuremberg, 2003

Steve Landers and Jean-Claude Wippler (2002)
Critcl: Beyond Stubs and Compilers
Ninth Tcl Conference, Vancouver, 2002

A. Markus (2002)
Combining Fortran and scripting languages
ACM SIGPLAN Fortran Forum, Volume 21, issue 3, 2002

Michael Metcalf and John Reid (1999)
Fortran 90/95 explained,
Oxford University Press, 1999, second edition

Michael Metcalf, John Reid and Malcolm Cohen (2004)
Fortran 95/2003 explained,
Oxford University Press, 2004

Appendix: A summary of Fortran 95

Fortran 95 is much more modern language than FORTRAN 77 (note the difference in upper and lower case), although it is still compatible with the older standard. Features have been added in the major revision known as Fortran 90 and later in the current standard Fortran 95 that take advantage of modern optimisation techniques, that enable a better organisation of the program code and many others. In this appendix I summarise the aspects I find most important:

- program organisation
- data types
- array handling

(For brevity, I will use the term "Fortran" in the rest of this appendix, to mean the current standard.)

Program organisation

Perhaps the greatest innovation is the concept of a "module". Modules in Fortran can be regarded more or less as classes in C++ or Java even though they do not define "object classes". Modules provide a large number of facilities:

- they can be used instead of the ill-reputed COMMON blocks to share data in a much more controlled way.
- they can contain subroutines and functions that are only accessible to the rest of the program by explicitly "using" the module. Thus modules provide a "namespace".
- subroutines and functions in a module automatically expose their interface, so that the compiler can check it against the actual call. Because this process is completely automatic (no manually maintained header files) it is much less error-prone than C's prototype mechanism. (The module that is part of Ftcl defines among other things interfaces to the C routines - in a very similar way as a C header file: as the C routines can not be part of a module, this way the compiler can check the calls.)

Modules do have a number of drawbacks:

- they can not cyclically depend on each other. Sometimes this requires a careful design.
- you can get a so-called compilation cascade: when large parts of a program depend on a particular module, changes in that module require recompilation of these parts of the program as well.
- the module intermediate files produced by the compiler are compiler-specific.

Fortran 90 added the so-called free-form for source code. Besides the old fixed-form (where the first six columns of a source file are reserved), Fortran now allows code to start anywhere and continuation is not achieved by a non-space character in the sixth position but via a trailing ampersand on the line that is to be continued.

Syntactical changes and the addition of a "select" statement have greatly reduced the need for statement numbers:

```
      do 110 i = 1,10
         ...
  110 continue
```

can now be written as:

```
do i = 1,10
   ...
enddo
```

(Only if you want to jump to a completely different part of the subprogram, like for error-handling, is a goto statement and a statement number necessary.)

Data types

While FORTRAN 77 only allowed primitive data types, integers and reals and so on, in Fortran 95 you can define so-called derived types, comparable to C's structs. If you want to, you can define operators to work on these new data types, for instance to control what happens if you assign derived types that contain pointers. With derived types you can achieve the same thing as with C's structs: a better organisation of the program's data.

Two other aspects of data types should be mentioned:

- "kinds" are a mechanism to select the range and the precision of a real or integer in an organised way:

  ```
  integer, parameter :: single_precision = kind(1.0)
  integer, parameter :: double_precision = kind(1.0d0)
  integer, parameter :: wp = single_precision ! Select a working
                                              ! precision "single"
  real(kind=wp) :: result
  ```

  Instead of replacing "real" by "double precision" in the program text, you now only need to change the value of the parameter "wp".

- attributes that make a variable allocatable or a pointer:

  ```
  real, allocatable, dimension(:), target :: array
  real, pointer, dimension(:)             :: p

  allocate( array(1:100) )
  array = 1.0        ! Set the entire array to 1.0
  p => array(21:30)  ! Associate p with a part of the array
  p = 2.0            ! Assign a new value to that part
  ```

  Unlike C, pointers can only point to variables that have the target attribute (or are pointers themselves) and syntactically they behave in much the same as ordinary variables.

Array handling

In many computational programs arrays are the principle type of data structure and iterations over all the elements of these arrays are very common. In Fortran these iterations can often be written by means of "array operations":

In the fragment below:

```
real, dimension(1:100) :: array
real                   :: sum_positive
integer                :: i

sum_positive = 0.0
do i = 1,size(array)
   if ( array(i) .gt. 0.0 ) then
      sum_positive = sum_positive + array(i)
   endif
enddo
```

the do-loop can be replaced by:

```
sum_positive = sum( array, array .gt. 0.0 )
```

where the "sum" intrinsic function has been used with a "mask", a logical expression to select particular elements of the array.

Array slices can be used to select a subset of the array elements too:

```
  total_even = sum( array(2:100:2))
```

If you use array operations (or the array control structures, "where" and "forall"), instead of explicit do-loops, the compiler can often do a better job at optimising that part of the program - one of the main goals of this language feature. A pleasant side effect is that the reader as well as the programmer have an easier job too. To illustrate that, the following statements reverse the order of the array elements:

```
   n = size(array)
   array(n:1:-1) = array
```

Doing that in an explicit loop can be tricky:

```
   n = size(array)
   do i = 1,n/2
      tmp = array(i)
      array(i) = array(n-i+1)
      array(n-i+1) = tmp
   enddo
```

The last feature I would like to mention is the allocation and deallocation of arrays (or of any variable) and the use of pointers:

```
   integer, dimension(:), pointer             :: p
   integer, dimension(:), allocatable, target :: array

   allocate( array(1:100) )
   ...
   if ( allocated( array ) ) then
      p => array(2:10)         ! Associate the pointer with a part only
   else
      ...
   endif

   if ( associated( p ) ) then
      write(*,*) 'p associated successfully'
   endif

   deallocate( array )
```

The functions associated() and allocated() allow the programmer to check the status of a pointer or an allocatable variable.

Final remarks

While Fortran 95 is a much more expressive language than its predecessors, it is also more difficult to use these new features efficiently and correctly. Still, with a few guidelines it can be put to good use, even if you do not have a master's degree in computing science.