MPPL to Fortran 90

Lee Taylor

March 15, 2002

UCRL-PRES-149697

Motivation

'I don't know what the language of the year 2000 will look like, but it will be called Fortran.' C.A.R. Hoare.

- Make the code you edit the same as the code you compile and debug. (line numbers match)
- Have compiler check as much as possible.
- Use the full features of f90.
- Prepare for Fortran 2000 (FMMV)
- Remove dependence on CRAY (integer) pointers
- Easier to import "foreign" packages.
- Easier to export "native" packages.
- Lower "buy in" for potential users.
- Reduce maintenance of MPPL.

New MPPL Flags

Several new flags have been added to access new functionality. The default behavior is still the same.

- --langf77 : convert language macros to f77 (default)
- --langf90 : convert language macros to f90
- --nolang : Do not convert language macros
- --isf90 : do not convert any language macros but still pretty print
- --pretty : Indent nicely (default)
- --nopretty : Do not change any indention
- --nonumeric : Leaves type declarations and constants alone. Does not read mppl.std.

More New MPPL Flags

- --continuation-indention (-ci) default is 3
- --comment-indention (-comi) default is 40
- --linelength : same as -l
- --relationalf77 : generate f77 relationals (default)
- --relationalf90 : generate new "C" style relationals
- --honor-newlines : preserve existing line break (assumed by --nopretty)

Process

New variable in config and Package file: MPPL_lang

- to f77 :Current default. Converts .m to .f
- to f90 : Converts .m to .f90. Uses --langf90. Sets compiler to use free form input.
- is f90 : Still does other macro and numeric processing.

Set in config file, can be overridden in Package file.

Process continued

- Set 'MPPL_lang = to f90' in config file. Now all generated files are .f90 Nothing in repository is changed, only generated files.
- Convert a package at a time using mppl --langf90 --nomacro --nonumeric --nopretty -l78
- Set Package 'MPPL_lang = is f90'
- Finish a package at a time.
- Set 'MPPL_lang = is f90' in config file. Remove from Package files.

Process questions

- Use .f or .f90 suffix?
- When .f files in repository are converted to free form, change to .f90 suffix?
- Order of package conversion?
- Speed of conversion?

Source Form

MPPL

- ; is a logical newline
- # and ! begin comments
- Automatic continuation if line ends in +, -, *, comma, (, &, |, ~, =, >, <
- White space is significant

Fortran 90 - free form input

- ; is a logical newline
- ! begin comments - C in column 1 is only in fixed format
- continuation - trailing ampersand, plus (in a few cases) leading ampersand on next line is required.
- White space is significant

Comments

- Input

```
    a = 1.     # initialize
```

- mppl --langf77

```
c initialize
    a = 1.
```

- mppl --langf90 --nopretty

```
    a = 1.     ! initialize
```

- mppl --langf90

```
    a = 1.                          ! initialize
```

- mppl --langf90 -comi25

```
    a = 1.             ! initialize
```

Continuations

Usual continuation

```
a = foo + bar + blat + &
    baz
```

An ampersand must be used on the continuing line if a keyword or character string is split between lines.

```
a = "The quick brown fox &
    &jumped over the lazy dogs back"
```

Clearer?

```
a = "The quick brown fox " // &
    "jumped over the lazy dogs back"
```

A statement may not have more than 40 lines.

Include

MPPL include. The file is passed thru MPPL for futher macro processing and the result is inserted into the output file.

```
include file
```

Fortran 90 include. MPPL never sees the contents of the file.

```
include "file"
```

End subroutine

Fortran 90 allows additional syntax on end statements.

```
subroutine foo

end subroutine foo
```

MPPL --langf90 generates this style end statement.

Logical operators

Fortran 90 supports "C" style relational operators

MPPL	F90	F77
>	>	.gt.
>=	>=	.ge.
<	<	.lt.
<=	<=	.le.
<> ~=	/=	.ne.
= ==	==	.eq.
&		.and.
|		.or.
~		.not.

Gotcha, not equal is /=, not !=

!= is a comment

Conversion will use F77 operators

If statements

No more magical continuations

- Input

```
    if (foo < 0)
        foo = 0.0
```

- mppl

```
    if (foo .lt. 0) foo = 0.0
```

- mppl -hnl

```
    if (foo .lt. 0)
   &    foo = 0.0
```

- mppl -hnl --langf90

```
    if (foo .lt. 0) &
        foo = 0.0
```

Do Loop

- Input

```
      do i=1,10
         j = j + 1
      enddo
```

- mppl --langf77

```
      do 23000 i=1,10
         j = j + 1
23000 continue
```

- mppl --langf90

```
      do i=1,10
         j = j + 1
      enddo
```

Until Loops

- Input

```
do
   i = i + 1
until (i==10)
```

- mppl --langf77

```
23000 continue
         i = i + 1
c until(i==10)
      if ( .not. (i.eq.10) )go to 23000
```

- mppl --langf90

```
do
   i = i + 1
   if (i.eq.10)exit
enddo
```

While Loops

- Input

```
      while (j < 10)
         j = j + 1
      enddo
```

- mppl --langf77

```
c while(j < 10)
23000 if (j .lt. 10) then
         j = j + 1
         go to 23000
      endif
c endwhile
```

- mppl --langf90

```
      do while (j .lt. 10)
         j = j + 1
      enddo
```

For Loops

- Input

```
for (i=0,i<10,i=i+1)
   print *, "hi"
endfor
```

- mppl --langf77

```
c -- for([i=0,i<10,i=i+1])
      i=0
      go to 23000
23001 continue
         i=i+1
23000    if (.not.(i.lt.10)) go to 23002
         print *, "hi"
c -- repeat
      go to 23001
23002 continue
c endfor
```

For Loops continued

- Input

```
for (i=0,i<10,i=i+1)
   print *, "hi"
endfor
```

- mppl --langf90

```
i=0
do while (i.lt.10)
   print *, "hi"
   i=i+1
enddo
```

Break and Next

- Input

```
do
   print *, "hi"
   break
   next
enddo
```

- mppl --langf77

```
23000 continue
         print *, "hi"
         go to 23001
         go to 23000
c -- repeat
      go to 23000
23001 continue
```

- mppl --langf90

```
do
   print *, "hi"
   exit
   cycle
enddo
```

Nested Next and Break

- Input

```
do
   do
      print *, "hi"
      break 2
      next 2
   enddo
enddo
```

- mppl --langf77

```
23000 continue
23002    continue
            print *, "hi"
            go to 23001
            go to 23000
c -- repeat
         go to 23002
c -- repeat
      go to 23000
23001 continue
```

Nested Next and Break continued

- mppl --langf90

```
      do
         do
            print *, "hi"
            go to 23001
            go to 23000
         enddo
23000    continue
      enddo
23001 continue
```

- "correct" way

```
outer: do
         do
            print *, "hi"
            exit outer
            cycle outer
         enddo
      enddo outer
```

Select

- Input

```
select(icase)
case 100, 101, 102, 103, 104, 105, 106:
   call other100
default: call warn
endselect
```

- Input

```
select(icase)
case 1: call foo1
case 4,7-10: call other
case 100:
   call other100
default: call warn
endselect
```

Select

- mppl --langf77

```
c select
      i23000= icase
         go to 23000
23002 continue

         call other100
23003    go to 23001
c -- case (default)
23004 continue
         call warn
c -- dispatch area for select
23005 go to 23001
23000 continue
      i23000=i23000-99
      if (i23000.lt. 1 .or. i23000.gt.7) go to 23004
      go to (23002,23002,23002,23002,23002,23002,23002), i23000
23001 continue
c endselect
```

Select

- mppl --langf77

```
      i23000= icase
         go to 23000
23002 continue
         call foo1
23003    go to 23001
23004 continue
         call other
23005    go to 23001
23006 continue
         call other100
23007    go to 23001
23008 continue
         call warn
23009 go to 23001
23000 continue
      if ( i23000 .eq. 1) go to 23002
      if ( i23000 .eq. 4) go to 23004
      if ( i23000 .ge. 7 .and. i23000 .le. 10) go to 23004
      if ( i23000 .eq. 100) go to 23006
      go to 23008
23001 continue
```

Select

- Input

```
select(icase)
case 100, 101, 102, 103, 104, 105, 106:
   call other100
default: call warn
endselect
```

- mppl --langf90

```
select case (icase)
case (100,101,102,103,104,105,106)
   call other100
case default
   call warn
end select
```

Select

- Input

```
select(icase)
case 1: call foo1
case 4,7-10: call other
case 100:
   call other100
default: call warn
endselect
```

- mppl --langf90

```
select case (icase)
case (1)
   call foo1
case (4,7:10)
   call other
case (100)
   call other100
case default
   call warn
end select
```

Select

Integer, logical and character variables are allowed in f90 case statements

- character

```
select case (style)
  case default
    call solid(x1,y1,x2,y2)
  case ("DOTS")
    call dots(x1,y1,x2,y2)
  case ("DASHES")
    call dashes(x1,y1,x2,y2)
end select
```

- logical

```
limit: select case (x > x_max)
   case (.true.)
     y = x * 0.9
   case (.false.)
     y = 1.0 / x
   end select limit
```

Return

Return assignment must be explicit

- Input

```
function foo(arg)
return(arg+1)
end
```

- mppl

```
function foo(arg)
foo = arg+1
return
end
```

- f90 result clause

```
function foo(arg) result (bar)
bar = arg+1
return
end
```

F90 Obsolete Features

- Arithmetic IF
- Assigned GOTO, assigned format
- Alternate RETURN
- PAUSE statement
- H edit descriptor

F90 Deprecated Features

- EQUIVALENCE statement
- COMMON statement
- BLOCK DATA statement
- ENTRY statement
- Fixed Form
- Double precision - Use Real([KIND=]kind)
- Computed GO TO
- Character length specification *len - Use CHARACTER([LEN=]len)
- Statement Functions

Next Phase
things to think about

F90 numeric model

- F90's syntax is the same as MPPL's.
- At least two approximation methods, one for default real and one for double precision real type, must be available.
- A processor may provide additional representation methods that may be declared using an explicit kind parameter.
- The values of the kind parameter are processor dependent. Basis uses the symbolic names Size2, Size4, Size8, and Size16.
- There are as many complex kinds are there are real kinds. (double complex is now standard)
- real*8 is not part of standard.

MPPL numeric processing

- Input

```
real foo1
real(Size4) foo2
real(Size8) foo3
foo1 = 1.0
foo2 = 1.0_Size4
foo3 = 1.0_Size8
```

- mppl -r4

```
real foo1
real foo2
doubleprecision foo3
foo1 = 1.0
foo2 = 1.0
foo3 = 1.0d0
```

- mppl -r8

```
doubleprecision foo1
real foo2
doubleprecision foo3
foo1 = 1.0d0
foo2 = 1.0
foo3 = 1.0d0
```

MPPL numeric processing

- Input

```
real foo1
real(Size4) foo2
real(Size8) foo3
foo1 = 1.0
foo2 = 1.0_Size4
foo3 = 1.0_Size8
```

- Proposed

```
real(RealD) foo1
real(Real4) foo2
real(Real8) foo3
foo1 = 1.0_RealD
foo2 = 1.0_Real4
foo3 = 1.0_Real8
```

- This assumes a generated file that is used by all routines.

```
integer, parameter :: Real4 = 4
integer, parameter :: Real8 = 8
integer, parameter :: RealD = Real4 or Real8
```

- default size is set based on -r4 or -r8 flag on a per package basis.

Address

- Input
  ```
  Address fwa
  ```
- mppl on sun
  ```
  integer fwa
  ```
- mppl on alpha
  ```
  integer*8 fwa
  ```
- Proposed
  ```
  integer(SizeA) fwa
  ```

Where the value of SizeA is generated to be the integer kind the same size as an address.

```
integer, parameter :: Integer4 = 4
integer, parameter :: Integer8 = 8
integer, parameter :: SizeA = Integer4 or Integer8
```

Character macros

- Input

```
Filename foo
Varname bar
```

- mppl

```
character*(256) foo
character*(129) bar
```

- Proposed

```
character(len=FILENAMESIZE) foo
character(len=VARNAMESIZE) bar
```

Numeric Summary

- Use types in the f90 standard way (non-deprecated)
- Single source for all architectures is possible
- SizeX cannot be used for integer and real since they are different kinds (Real4 and Integer4 instead of just Size4)
- This only affects the generated code (for now).

MPPL and Mac

Mac generates lots of MPPL macros.

Typical VDF file

```
xxx
define MAXSIZE 10.0
*** Stuff :
bar real
```

Generates macxxx (a.k.a. .d) file

```
define MAXSIZE 10.0
define([UseStuff],[Remark([ Group Stuff])\
      double precision bar
      common /xxx03/ bar
Remark([ End of Stuff])\
])
```

MPPL and Mac

Generated macxxx

```
define MAXSIZE 10.0
define([UseStuff],[Remark([ Group Stuff])\
      double precision bar
      common /xxx03/ bar
Remark([ End of Stuff])\
])
```

MPPL source

```
      subroutine init
UseStuff
      bar = MAXSIZE
      end
```

MPPL macxxx init.m

```
      subroutine init
c Group Stuff
      double precision bar
      common /xxx03/ bar
c End of Stuff

      bar = 10.0
      end
```

MPPL and Mac

Typical VDF file

```
xxx
define MAXSIZE 10.0
*** Stuff :
bar real
```

Proposed

- One file per package
- One file per group

xxx.inc

```
        include "kinds.inc"
        real(RealD), parameter :: MAXSIZE = 10.0
```

stuff.inc

```
! Group Stuff
        integer, real(RealD) :: bar
        common /xxx03/ bar
! End of Stuff
```

MPPL and Mac

Create a directory structure under include to hold generated include files.

```
dev/ARCH/include
                xxx
                   xxx.inc
                   stuff.inc
```

- Avoids large number of files at include level
- Controls name space better to avoid conflicting names

MPPL and Mac

Convert Use macros to include statements.

MPPL source

```
        subroutine init
   UseStuff
        bar = MAXSIZE
        end
```

converted source

```
        subroutine init
        include "xxx.inc"
        include "stuff.inc"

        bar = MAXSIXE
        end
```

Uses f90 include, not MPPL include so include files will not be processed by MPPL.

MAC issues

MPPL is case sensitive, F90 is not.

```
      maxsize = MAXSIZE
```

Embedded macros in VDF file

```
xxx
define MAXSIZE 10.0
*** Stuff :
%define MINSIZE 0.0
bar real
```

Generates

```
define MAXSIZE 10.0
define([UseStuff],[Remark([ Group Stuff])\
define MINSIZE 0.0
      double precision bar
      common /xxx03/ bar
Remark([ End of Stuff])\
])
```

Mac and Modules

Another option is to generate a file with a module for each group.

Typical VDF file, note language attribute

```
xxx
define MAXSIZE 10.0
*** Stuff language "F90":
bar real
```

xxx_mod.f90

```
      module xxx_mod
      include "kinds.inc"
      real(RealD), parameter :: MAXSIZE = 10.0
      end module xxx_mod

      module stuff_mod
! Group Stuff
      integer, real(RealD) :: bar
      end module stuff_mod
```

Notice, no common block.

Mac and Modules

MPPL source

```
        subroutine init
   UseStuff
        bar = MAXSIZE
        end
```

Converted

```
        subroutine init
        use xxx_mod
        use stuff_mod

        bar = MAXSIXE
        end
```

Mac and Modules issues

- One source file but still compiles to one .mod (or .M) file per module. Still need a package directory level.
- Introduce more dependencies since module must be compiled before it can be USEd.
- Totalview has some issues with module variable. Sometimes will not "dive" on them.
- Using language attribute, possible to use only where wanted.

Internal functions

```
subroutine foo
   a = 1
   b1 = bar(2.0)
   b2 = bar(3.0)
contains
   function bar(b)
      bar = a + b
   end subroutine bar
 end subroutine foo
```

- An internal procedure definition cannot have an internal procedure part.
- The default type mapping in an internal procedure is the type mapping of the host.
- Replaces statement functions

Internal functions and macros

- Using a macro

```
define(setup,[
      a = $1 + $2
])

      subroutine foo
          real a
          setup(1,2)
      end subroutine foo
```

- Using an internal procedure

```
      subroutine worker
          real a
          setup(1,2)
      contains
          subroutine setup(x,y)
            a = x + y
          end subroutine setup
      end subroutine worker
```

Variable 'a' in setup is same as variable 'a' in worker.

Future

Most files will not require MPPL at all. Some conditional compilation or macro processing may still be needed to overcome compiler or site features.

Once the code is f90 we can use third-party tools to help with other conversions.

- Implicit none
- Array syntax
- Explicit Interfaces
- Modules
- F90 pointers

Tools - Fortranlint

Cleanscape FortranLint is a Fortran static source code analysis tool that reduces your organizational exposure to risks from latent software problems by automatically identifying problems at their source -- in the Fortran code prior to compiling or executing programs. From its first use, this venerable Fortran source code analysis tool can save you hundreds of hours in Fortran code debugging, greatly reducing resources required for Fortran testing efforts.

http://www.cleanscape.net/products/fortranlint

Tools - VAST/77to90

With VAST/77to90 you can easily update all your existing Fortran 77 programs to clean and efficient Fortran 90. This is much more than a source form reformatter. For example, VAST/77to90's sophisticated Fortran 90 capability provides automatic generation of multi-dimensional array syntax.

- Removal of obsolete features
- Elimination or reduction of GOTOs and labels
- Generation of array syntax in place of loops.
- Creation of MODULEs from COMMONs
- Automatic generation of interface blocks.
- Fortran "lint" diagnostics

http://www.psrv.com/vast77to90.html

Tools - simulog

Automatically convert Fortran 77 code to Fortran 90 Use the restructuring tool to convert old code to the new syntax:

- spaghetti code converted to structured constructs.
- new declaration syntax, all variables declared
- common blocks converted to modules
- automatic interface block generation. Even F77 proprietary extensions can migrate, such as Cray-style pointers converted to F90 pointers.

Use FORESYS on mixed style F90/F77 applications If you decide to keep some parts of your application coded in F77, FORESYS can still be used to ensure top quality with the analysis tool.

http://www.simulog.fr/is/fore3.htm

Tools - plusFort

plusFORT Version 6 is unique among QA tools in offering three distinct and complementary approaches to the problem of software quality assurance. Working together these three approaches have far more impact than any one could by itself.

- Static Analysis - views data usage from a global perspective, and detects errors and anomalies that compilers and other tools miss.
- Dynamic Analysis - Calls to probe routines are inserted in the source code before any operation which depends on the value of a data item, and the program is compiled and linked in the normal way
- Test Coverage and Hot-Spot Analysis - places probes into Fortran source code which allow users to monitor the effectiveness of testing.

http://www.polyhedron.com/pf/pfqa.html

Notice

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Notice continued

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information

P.O. Box 62, Oak Ridge, TN 37831
Prices available from (423) 576-8401
http://apollo.osti.gov/bridge/

Available to the public from the National Technical Information Service

U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161
http://www.ntis.gov/

OR

Lawrence Livermore National Laboratory
Technical Information Department's Digital Library
http://www.llnl.gov/tid/Library.html