Modern Fortran: Meta-programming with Fypp

Bálint Aradi

2018 Workshop on Fortran Modernization for Scientific Application (ForMoSA)

Outline

Going beyond Fortran

- **Meta-programming with the Fypp pre-processor**

Why is modern Fortran not enough?

Fortran language still misses some important capabilities

- Conditional compilation
- Generic programming techniques (a.k.a. templates)

Conditional compilation

Parts of the code should be compiled only under certain conditions

- **Optional external libraries** (e.g. mathematical libraries ...)

Lapack is not present during build

```
subroutine solve(aa, bb)
  ! Hand coded algorithm
  …
end subroutine solve
```

Lapack is present during build

```
subroutine solve(aa, bb)
  ! Invoke LAPACK routine
  call dgesv(aa, bb, …)
end subroutine solve
```

- **Different build types** (e.g. debug vs. production, serial vs. MPI-parallel)

Debug build (include consistency checks)

```
subroutine solve(aa, bb)
! Make consistency checks
…
! Do actual algorithm
…
```

Discard expensive checks

```
subroutine solve(aa, bb)
! Do actual algorithm
…
```

Alternative to conditional compilation

- You handle optional components on the Fortran level
- Define **stub functions** / **data types** if optional component is not present
- **Link** those functions / modules to your application, so that routine names can be resolved by the linker (although they should be never called)

```
subroutine solve(aa, bb)
  if (hasLapack) then
    call dgesv(aa, bb, …)
  else
    ! Do hand coded way
end subroutine solve
```

```
subroutine dgesv(aa, bb, …)
  write(error_unit, *) &
    & 'Internal error:&
    & dgesv called despite&
    & missing LAPACK'
  stop
end subroutine solve
```

Stub function used if built without LAPACK

Disadvantages:

- Mistakes in code-flow detected during **run-time**
- Tedious if optional component provides **many routines**
- Tedious if optional component provides **complex data types** (evtl. with type bound procedures)

Conditional compilation tools

CoCo (Conditional Compilation)

- It **was** part of the Fortran standard, but has been officially withdrawn
- Open source (GPL) implementation (in Fortran!) exists
- Uses Fortran-like preprocessor language with **??** as prefix

```
?? logical, parameter :: has_intrinsic_module = .false.
program hello_world
?? if (has_intrinsic_module) then
use, intrinsic :: iso_fortran_env, only: output_unit
write (unit=output_unit, fmt= *) 'Hello, world!'
?? else
write (unit= *, fmt= *) 'Hello, world!'
?? endif
end program hello_world
```

Original example on Dan Nagle's Technical Site

Conditional compilation tools

C-preprocessor: cpp

- Using (abusing) the **pre-processor of the C / C++ languages**
- It may misinterprete language constructs if not used carfully (e.g. character concatenation operator // is a comment in C++)
- Output may not be 100% Fortran compatible (e.g. line length)

Fortranized C-preprocessor: fpp

- Same syntax as cpp but (mostly) without C/C++ artifacts
- Bundled (in some form) with most compilers (GNU, Intel, NAG, …)
- Used by most many Fortran projects (either fpp or raw cpp)

```
#ifdef WITH_EXTERNAL_LIBRARY
call external_library_function()
#else
call hand_code_version()
#endif
```

Conditional compilation tools

Various open-source Fortran pre-processors

- f90ppr, Forpedo, Fpx3, PyF95++, PreForM.py, Fypp, ufpp, …
- See the Preprocessor page of the Fortran Wiki for some details
- Implemented in various languages, mostly Fortran and Python
- Their functionality and syntax differ (some of them cpp compatible, though)
- Level of documentation and support vary on a very broad scale

Q Which pre-processor should I use for my Modern Fortran project?

A1 **None**, just stick to the Fortran standard (**the safe bet**)

A2 If you need **conditional compilation only**, take **fpp** as it is used by the majority of the Fortran projects (**principle of least surprise**)

A3 At some point, you may need **meta-programming** capabilities, so let’s investigate further …

Meta-programming

Fortran “enriched / degraded” with additional language constructs

- Code is not valid Fortran any more
- **Pre-processor** must be invoked before the compilation in order to transform the code into standard conforming Fortran code
- Strictly speaking: conditional compilation is alredy meta-programming
- Other languages (e.g. C++) provide standardized built-int tools (e.g. C++ templates)

Q Do I really need this meta-mambo-jumbo for my Fortran project?

A1 **No!** Already Fortran 77 offered all features a scientist / engineer ever needs. Everything since then just made Fortran more complicated and error prone without gaining anything...

A2 **Yes**, you do! Let’s **investigate a trivial example** (**proof of concept**)

A trivial meta-programming task

Swap the content of two variables of the same type, rank and shape

integer, rank 0 version

```
subroutine swap(aa, bb)
  integer, intent(inout)&
      & :: aa, bb

  integer, allocatable :: cc

  cc = aa
  aa = bb
  bb = cc

end subroutine swap
```

real, rank 0 version

```
subroutine swap(aa, bb)
  real, intent(inout)&
      & :: aa, bb

  real, allocatable :: cc

  cc = aa
  aa = bb
  bb = cc

end subroutine swap
```

Any striking similarities between the two versions?

- Algorithms exactly the same, **only data type changes**

A trivial meta-programming task

- Exactly the same algorithm also works for multi-dimensional arrays (Fortran rocks!)

real, rank 2 version

```
subroutine swap(aa, bb)
  real, intent(inout) :: aa(:,:), bb(:,:)

  real, allocatable :: cc(:,:)

  cc = aa
  aa = bb
  bb = cc

end subroutine swap
```

- Again **only data type (and rank) changed**

A trivial meta-programming task

- Abitrary derived types (or any arrays thereof) should be no problem either

derived type, rank 0 version

```
subroutine swap(aa, bb)
  type(MyType), intent(inout) :: aa, bb

  type(MyType), allocatable :: cc

  cc = aa
  aa = bb
  bb = cc

end subroutine swap
```

- And so on and so on …
- It would be nice, if we had not to repeat the same code again and again (**avoiding code duplication**)

Why not just repeat the code with all it types it is needed for?

- **Error-prone**
  → *Are you sure you fixed the recent bug in all 27 duplicates in your code?*

- Painfully **complicated to share** with others
  (when coding open source it's all about **sharing & colaborating**)

```
In order to use our generic library in your code, you have to
carry out the following steps:

1. copy the interesting parts of the library into your code
2. find and replace all relevant types (marked by special
   markers) with the type you want to use it for.

You will have to repeat the procedure for every type you need
and every time you download an updated version of the library.

We wish you good luck and a lot of fun!

The Code Repeaters
```

The (standard conforming!) trick with include

- Generic routines act on an **undefined type with given name**
- In the specific implementation the **type** with the given name is **defined** first, then the **generic routines** are **included**.

generic.inc

```
subroutine swap(aa, bb)
  type(MyType), … :: aa, bb
  type(MyType), … :: cc
  …
end subroutine swap
```

specific.f90

```
type :: MyType
  …
end type MyType
…
include 'generic.inc'
…
```

Some disadvantages:

- The generic file is not self-contained (can not be compiled on its own)
- Only works with derived types, not with intrinsic types (integer, real, …)
- Generic file must contain a separate implementation for each rank

Using a pre-processor

- Define a **macro** which contains the **generic implementation** with the type, rank, etc. as variables
- In the **specific implementation** expand the macro with the specific type name, rank, etc.
- Pass also all extra information needed to create the specific implementation via macro arguments (or pre-processor) variables

Which preprocessor ?

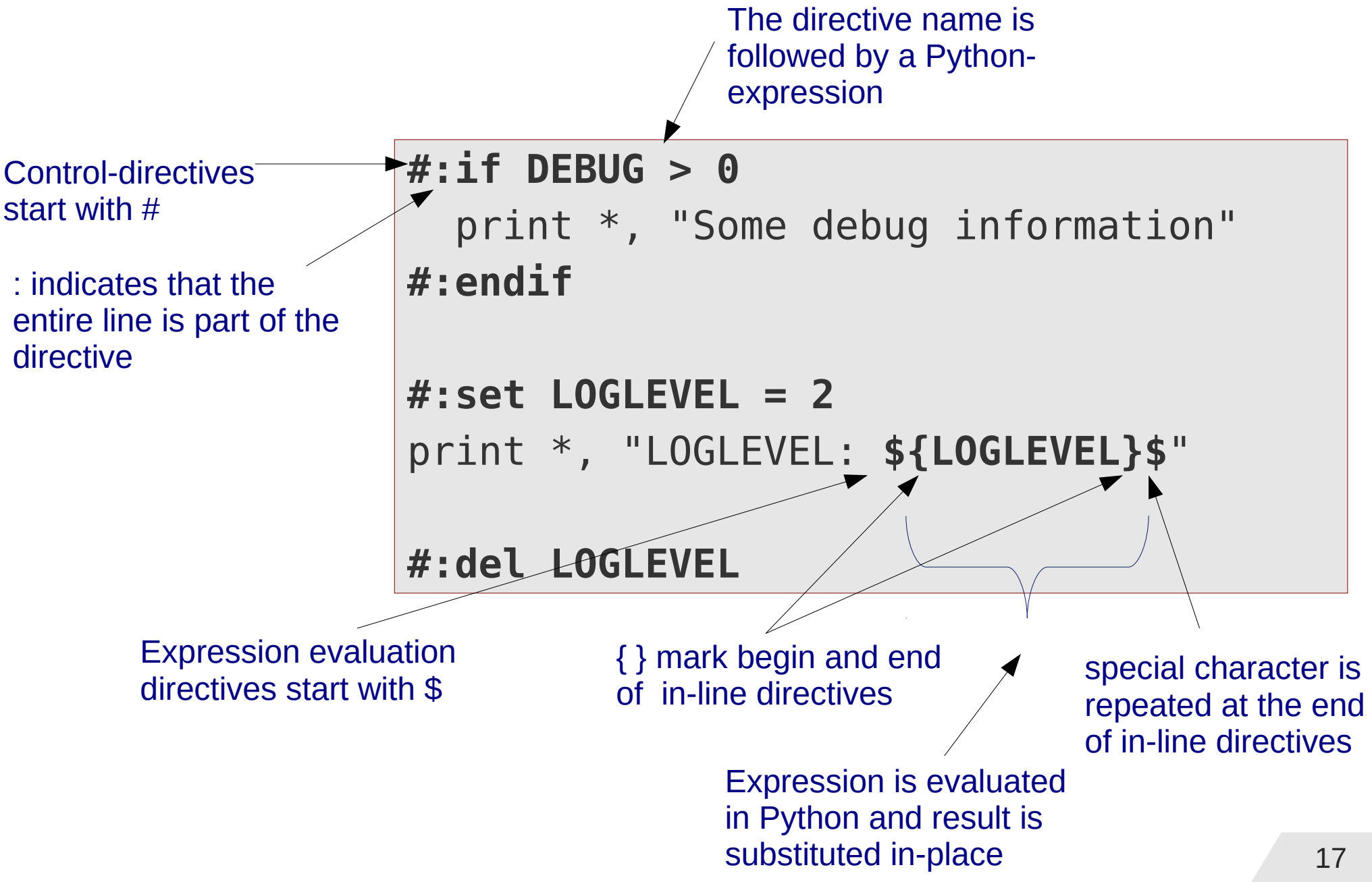
- Theoretically most pre-processors (even fpp / cpp) can be used
- I think, **Fypp** can solve it more elegant and compact than the others
- Especially the code still looks quite pretty and clean after it had been pre-processed (easy to debug the specific implementation)

NOTE: Do not believe me! Being the main author of Fypp, I may be biased a lot. Try the examples and the exercises with both, Fypp and your favorite pre-processor and judge yourself.

General design principles

- **Simple**, easy to use pre-processor
- Emphasis on **robustness and neat integration** into Fortran developing toolchains and workflows.
- Avoid to create yet another mini-language for expression evaluation
  - → **Use Python expressions**
- Make it **easy to incorporate** it in your project
  - → One file only (just bundle this file with your project to avoid having Fypp as prerequisite)
- Break compatibility with the C-preprocessor (no reason to replicate all issues one gets when using cpp / fpp with Fortran), but look nevertheless similar to enable a quick learning curve.
- **Minimalistic on features**
  - → It should not tempt you to do something with Fypp which can be done in Fortran with reasonable efforts

Definition, evaluation and removal of variables



Fypp – quick feature overview

Macro definitions and macro calls

- Defines a parameterized text block, which can be inserted

```
#:def ASSERT(COND)
#:if DEBUG > 0
if (.not. ${COND}$) then
  print *, "Failed: file ${_FILE_}$, line ${_LINE_}$"
  error stop
end if
#:endif
#:enddef ASSERT

$:ASSERT('size(myArray) > 0')

! Direct call (no quotation)
@:ASSERT(size(myArray) > 0)
```

Macro def. may contain further Fypp directives

Built-in pre-processor variables

Evaluation expression over entire line

Passing a Python string as argument

The macro is a Python callable, which returns its text content

More convenient calling form without the need for quotation

Fypp – quick feature overview

Conditional output

```
program test
#:if defined('WITH_MPI')
  use mpi
#:elif defined('WITH_OPENMP')
  use openmp
#:else
  use serial
#:endif
```

defined() is a Python function with a string argument

Returns True / False if a variable with the given name has / has not been defined

Iterated output (Fortran templates!)

Iteration variable

Arbitrary Python iterable

```
interface myfunc
#:for SUFFIX in ['real', 'dreal', 'complex', 'dcomplex']
  module procedure myfunc_${SUFFIX}$
#:endfor
end interface myfunc
```

Fypp – quick feature overview

Insertion of arbitrary Python expression

```
character(*), parameter :: comp_date = &
    & "${time.strftime('%Y-%m-%d')}$"
```

Note: The module must be imported with the command line option **-m**

Inclusion of files during pre-processing

```
#:include "macrodefs.fypp"
```

Fortran-style continutation lines in pre-processor directives

```
#:if var1 > var2 &
    & or var2 > var4
  print *, "Doing something here"
#:endif
```

Passing multiline string arguments to callables

```
#! Callable needs only string argument
#:def DEBUG_CODE(CODE)
  #:if DEBUG > 0
    $:CODE
  #:endif
#:enddef DEBUG_CODE

#! Code block passed as first positional argument
#:call DEBUG_CODE
  if (size(array) > 100) then
    print *, "Debug info: spuriously large array"
  end if
#:endcall DEBUG_CODE
```

Passing multiline string arguments to callables

```
#! Callable needs also non-string argument types
#:def REPEAT_CODE(CODE, REPEAT)
  #:for _ in range(REPEAT)
    $:CODE
  #:endfor
#:enddef REPEAT_CODE

#! Pass code block as positional argument
#! and 3 as keyword argument "REPEAT"
#:call REPEAT_CODE(REPEAT=3)
this will be repeated 3 times
#:endcall REPEAT_CODE
```

Fypp – quick feature overview

Preprocessor comments

```
#! This will not shown in the output
#! Newline chars at the end will be also suppressed
```

Suppressing the preprocessor output in selected regions

```
#:mute
#:include "macrodefs.fypp"
#:endmute
```

You can suppress all the newlines contained in the include file

Explicit request for stopping the pre-processor

```
#:if DEBUGLEVEL < 0
  #:stop 'Negative debug level not allowed!'
#:endif
```

Fypp – quick feature overview

Easy check for macro parameter sanity

```
#:def mymacro(RANK)
  #! Macro only works for RANK 1 and above
  #:assert RANK > 0
  :
#:enddef mymacro
```

Line numbering directives in output (with command line option -n)

```
program test
#:if defined('MPI')
use mpi
#:endif
```

→

```
# 1 "test.fypp"
program test
# 3 "test.fypp"
use mpi
# 5 "test.fypp"
```

and many more …

See the **Fypp-manual** for more examples and further details

Fypp – metaprogramming demo

Swap the content of two variables of the same type, rank and shape

- First we create a file with macro(s) useful for working with Fortran in general (Fypp is Fortran-agnostic)

fortrandefs.fypp

```
#:def RANKSUFFIX(RANK)
#:if RANK > 0
$:'(:' + ',:' * (RANK - 1) + ')'
#:endif
#:enddef RANKSUFFIX
```

Addition of strings

Repetition of strings

- **RANKSUFFIX(*RANK*)** returns
  - → **Empty string** if *RANK* is 0
  - → **(:)** if *RANK* is 1
  - → **(:,:)** if *RANK* is 2

  etc.

Fypp – metaprogramming demo

- Then we write the **generic template** for the swap-algorithm

swap.fypp

```
#:include 'fortrandefs.fypp'

#:def SWAP_ROUTINE_TEMPLATE(ROUTINE_NAME, TYPE, RANK)

subroutine ${ROUTINE_NAME}$(aa, bb)
  ${TYPE}$, intent(inout) :: aa${RANKSUFFIX(RANK)}$
  ${TYPE}$, intent(inout) :: bb${RANKSUFFIX(RANK)}$

  ${TYPE}$, allocatable :: cc${RANKSUFFIX(RANK)}$

  cc = aa
  aa = bb
  bb = cc
end subroutine ${ROUTINE_NAME}$

#:enddef SWAP_ROUTINE_TEMPLATE
```

Fypp – metaprogramming demo

- Finally, we create a module with all **specific implementations** we need

swap.F90

```
#:include 'swap.fypp'
#:set IMPLEMENTATIONS = [('dreal3', 'real(dp)', 3),&
    & ('int2', 'integer', 2)]

module swap_module
  use accuracy, only : dp
  interface swap
  #:for SUFFIX, _, _ in IMPLEMENTATIONS
    module procedure swap_${SUFFIX}$
  #:endfor
  end interface swap
contains
  #:for SUFFIX, TYPE, RANK in IMPLEMENTATIONS
  $:SWAP_ROUTINE_TEMPLATE('swap_' + SUFFIX, TYPE, RANK)
  #:endfor
end module swap_module
```

For simplifying code later, we create a list with the specific implementations needed

Dummy variables

Fypp – metaprogramming demo

- After **pre-processing** the file with the specific implementations we get

```
fypp swap.F90 swap.f90
```

```
interface swap
  module procedure swap_dreal3
  module procedure swap_int2
end interface swap
```

swap.f90

```
subroutine swap_dreal3(aa, bb)
  real(dp), intent(inout) :: aa(:,:,:)
  real(dp), intent(inout) :: bb(:,:,:)
  …
end subroutine swap_dreal3
…
subroutine swap_int2(aa, bb)
  integer, intent(inout) :: aa(:,:)
  integer, intent(inout) :: bb(:,:)
  …
end subroutine swap_int2
```

- Pre-processing should be done during the build process
- If the generic templates in swap.fypp are changed or updated, the build process automatically produces actualized specific implementations

Final Fypp notes

What Fypp intends to be

- A cool **pre-processor** using a cool language (Python) familiar to the majority of cool modern Fortran programmers
- A useful tool to **work around missing generic programming features** in Fortran
- Especially, it wanted to allow for **generic programming with static typing** (detecting type mismatches at compile time, instead of at run-time)

What Fypp never intended to be

- A tool for creating a Fortran-Python **hybrid language** (although interesting attempts do exist)
- A **long term solution** (although we will maintain and develop it for quite a while as already several projects – including our own – heavily rely on it).

Let's lay our trust in the Fortran Standard Committee that Fortran obtains generic programming functionalities soon making Fypp superfluous.

(Unfortunately, "soon" could mean decades in reallity – as of 2018 most compiler vendors have not finished the implementation of the Fortran 2008 standard yet)