Gfort2py

A fast and easy way to bridge between Python and Fortran

Rob Farmer

What do I mean?

```
subroutine foo(str)
    character(len=:) :: str

     write(*,*) str

end subroutine
```

gfort2py

```
>>> foo(“Hello World”)
Hello World
```

What’s the point?

- **Large existing code bases with useful code**
- **But,**
  - Can be hard to interact with
  - Experiment with and visualise output
- **Python’s REPL**
  - Very useful when your not quite sure what your doing

A little background

- https://github.com/rjfarmer/gfort2py
- ~9 years old
- Written in Python & Fortran
- GPL-2.0 or later
- Currently on v2.6.2
  - 3.0 release in the works
- Install
  - python -m pip install gfort2py

Design principles

- Minimal to no changes in Fortran code
- Minimal to no changes needed in build system
- Support as many new Fortran features as possible
- Supports only gfortran

How does this all work?

```
subroutine foo(str)
    character(len=:) :: str

     write(*,*) str

end subroutine
```

foo.f90

How does this all work?

```
module foobar
contains
   subroutine foo(str)
      character(len=:) :: str

      write(*,*) str

   end subroutine
end module
```

foo.f90

```
gfortran -fPIC -shared  foo.f90 -o foo.so
gfortran -dynamiclib foo.f90 -o foo.dylib
gfortran -shared foo.f90 -o foo.dll
```

foobar.mod
foo.so

How does this all work?

foobar.mod
foo.so

```
>>> import gfort2py as gf

>>> x = gf.fFort("foo.so","foorbar.mod")

>>> x.foo("Hello World")
Hello World
```

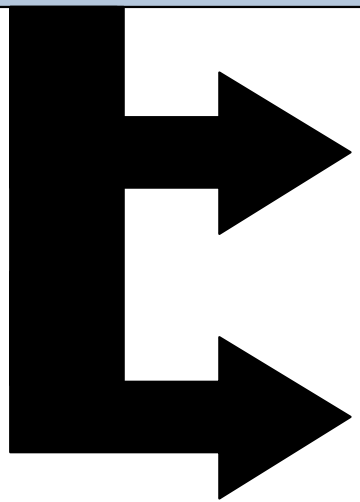
How does this all work?

```
module foobar
    real, parameter :: pi=3.14
    integer :: y
end module
```

```
>>> import gfort2py as gf

>>> x = gf.fFort("foo.so","foorbar.mod")

>>> print(x.pi)
3.14

>>> x.y = 1
```

What’s going on?

```
>>> x = gf.fFort(“foo.so”,”foorbar.mod”)
```



”foorbar.mod” → gfModParser → Understand the Fortran interface

“foo.so” → Python ctypes → Create C-interface

An aside on mod files

- Modules provide the interface to Fortran code
- Mod files are generated by the compiler and “save” the data
- Gfortran embeds everything you need to know to interact with a module

Why bother with mod files?

- **They are compiler specific**
- **Compiler version specific**
  - Well, ABI level
  - Gfortran changed at v15
  - Previously the same since v8
- **New versions of compiler need development work**

Benefits of using the mod file

Fortran source code can be hard to parse

What is the C equivalent?

```
integer :: x
```

What’s the equivalent C type?

Just use iso_c_binding and BIND(c)?
Only works if someone did that from the start

What is the C equivalent?

integer :: x

int

What is the C equivalent?

```
integer(kind=wk) :: x
```

??

What is the C equivalent?

```
integer, parameter :: wk = 8
integer(kind=wk) :: x
```

int64

Sure, we shouldn't just assume kind=8 is 8 bytes but advantage of fixing on one compiler is we can embed its assumptions

What is the C equivalent?

```
use module, only wk=> mykind
integer(kind=wk) :: x
```

??

What is the C equivalent?

```
use module, only wk=> mykind
integer(kind=wk) :: x
```

??

gfortran -lother -Iother ...

What is the C equivalent?

```
#ifdef FOO
    integer, parameter :: wk=4
#else
    integer, parameter :: wk=8
#endif

integer(kind=wk) :: x
```

??

Benefits of using the mod file

Lets also quietly ignore things like:

- -fdefault-integer-8
- -finteger-4-integer-8

Benefits of using the mod file

The compiler has done the hard work

Mod files resolve all the issue with understanding what a object is

At the cost of being tied to 1 compiler

The insides of a mod file

```
GFORTRAN module version '15' created from basic.f90
(() () () () () () () () () () () () () () () () () () () () () () () ()
() () ())

()

()

()

()

()

(2 '__convert_i4_i8' '(intrinsic)' '' 1 ((PROCEDURE UNKNOWN-INTENT
UNKNOWN-PROC UNKNOWN UNKNOWN 0 0 FUNCTION ELEMENTAL PURE
ARRAY_OUTER_DEPENDENCY) () (INTEGER 8 0 0 0 INTEGER ()) 0 0 () () 2 () ()
() 0 0)
3 '__convert_r4_i4' '(intrinsic)' '' 1 ((PROCEDURE UNKNOWN-INTENT
UNKNOWN-PROC UNKNOWN UNKNOWN 0 0 FUNCTION ELEMENTAL PURE
ARRAY_OUTER_DEPENDENCY) () (INTEGER 4 0 0 0 INTEGER ()) 0 0 () () 3 () ()
() 0 0)
4 'a_int' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC UNKNOWN
IMPLICIT-SAVE 0 0) () (INTEGER 4 0 0 0 INTEGER ()) 0 0 () () 0 () () ()
0 0)
5 'a_int_lp' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC UNKNOWN
IMPLICIT-SAVE 0 0) () (INTEGER 8 0 0 0 INTEGER ()) 0 0 () () 0 () () ()
0 0)
6 'a_int_lp_set' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC
UNKNOWN IMPLICIT-SAVE 0 0) () (INTEGER 8 0 0 0 INTEGER ()) 0 0 () () 0 ()
() () 0 0)
7 'a_int_mixed' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC
UNKNOWN IMPLICIT-SAVE 0 0) () (INTEGER 4 0 0 0 INTEGER ()) 0 0 () () 0 ()
() () 0 0)
8 'a_int_set' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC
UNKNOWN IMPLICIT-SAVE 0 0) () (INTEGER 4 0 0 0 INTEGER ()) 0 0 () () 0 ()
() () 0 0)
```

- Gfortran these are gzipped text files
- Gunzip < file.mod > file.txt

- Read a bit like lisp
- Look for opening “(“ and closing “)” for nesting level
- Has things like type, kind, name, array properties etc
- Contains (almost) everything you need

The insides of a mod file

```
GFORTRAN module version '15' created from basic.f90
(() () () () () () () () () () () () () () () () () () () () () () () ()
() () ())

()

()

()

()

()

(2 '__convert_i4_i8' '(intrinsic)' '' 1 ((PROCEDURE UNKNOWN-INTENT
UNKNOWN-PROC UNKNOWN UNKNOWN 0 0 FUNCTION ELEMENTAL PURE
ARRAY_OUTER_DEPENDENCY) () (INTEGER 8 0 0 0 INTEGER ()) 0 0 () () 2 () ()
() 0 0)
3 '__convert_r4_i4' '(intrinsic)' '' 1 ((PROCEDURE UNKNOWN-INTENT
UNKNOWN-PROC UNKNOWN UNKNOWN 0 0 FUNCTION ELEMENTAL PURE
ARRAY_OUTER_DEPENDENCY) () (INTEGER 4 0 0 0 INTEGER ()) 0 0 () () 3 () ()
() 0 0)
4 'a_int' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC UNKNOWN
IMPLICIT-SAVE 0 0) () (INTEGER 4 0 0 0 INTEGER ()) 0 0 () () 0 () () ()
0 0)
5 'a_int_lp' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC UNKNOWN
IMPLICIT-SAVE 0 0) () (INTEGER 8 0 0 0 INTEGER ()) 0 0 () () 0 () () ()
0 0)
6 'a_int_lp_set' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC
UNKNOWN IMPLICIT-SAVE 0 0) () (INTEGER 8 0 0 0 INTEGER ()) 0 0 () () 0 ()
() () 0 0)
7 'a_int_mixed' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC
UNKNOWN IMPLICIT-SAVE 0 0) () (INTEGER 4 0 0 0 INTEGER ()) 0 0 () () 0 ()
() () 0 0)
8 'a_int_set' 'basic' '' 1 ((VARIABLE UNKNOWN-INTENT UNKNOWN-PROC
UNKNOWN IMPLICIT-SAVE 0 0) () (INTEGER 4 0 0 0 INTEGER ()) 0 0 () () 0 ()
() () 0 0)
```

- Decoding this is non-trivial
- gfModParser
  - https://github.com/rjfarmer/gfModParser
  - Turns the mod file into something useful

Other missing piece

- **Some things need extra handling**
  - Hidden arguments for characters
  - Returning characters/arrays from functions
  - Optional arguments
  - Pass by value

Other missing piece

- **-fdump-tree-original**
  - Not needed at runtime but help full when developing
- **Generates a .original file**
- **Helps determine when you need a hidden argument(s)**

Other missing piece

```
character(len=5) function func_ret_str(x)

  character(len=5),intent(in) :: x
```

```
void func_ret_str (
character(kind=1)[1:5] & __result,
integer(kind=8) .__result,
character(kind=1)[1:5] & restrict x,
integer(kind=8) _x)
```

Function returns void

Return string and its length prepended to argument list

Input string and its length appended to argument list

How to do .....

Procedures

```
integer myfunc(x)

subroutine mysub(x)
```

```
>>> res = x.myfunc(1)
 Result(result=2, args={'x': 1})
>>>print(res.result)
2

>>> res = x.mysub(1)
 Result(result=None, args={'x': 1})
>>>print(res.args['x'])
```

“args” contains all arguments passed in/out

Arrays

```
! Supported
Integer :: a(5), a(N),a(*),a(:)
Integer :: a(myfunc(N)+1)
Integer, allocatable :: a(:)
```

```
>>> x.a = np.array([1,2,3.....])

>>> type(a)
numpy.ndarray

>>> x.myfunc(np.array([1,2,3.....]))
```

Python arrays start at index 0 while Fortran is at 1 (default)

Derived types

```
Type foo
   Integer :: x,y,z
End type

type(foo) :: bar
```

```
>>> bar['x'] = 1

>>> type(bar)
gfort2py.fDT

>>> x.myfunc(bar)

 >>> x.myfunc({'x':1,'y':2})
```

On the Python side derived types are dict-like objects

Arrays of derived types

```
! Supported
type(foo) :: bar(3,3)

! Not supported (yet)
type(foo), allocatable :: bar(:)
```

```
>>> bar[2,2]['x'] = 1

>>>print(bar[2,2]['x'])
1
```

Quadruple precision

```
real(real128) :: x
```

Python has no standard
128-bit float

Quadruple precision

```
real(real128) :: x
```

```
>>> import pyquadp as pq

>>> print(foo.x)
pyquadp.qfloat(1)
```

https://github.com/rjfarmer/pyQuadp

Wraps __float128, __complex128, and __int128,
As well as math routines (sin,cos, log etc) from libquadmath

Module free mode

```
>>> fortran = """
subroutine mysub()
....
end subroutine my sub()
"""
```

```
>>> x = gf.compile(string=fortran)

>>> x.mysub()
```

I'm just making a module for you in the background

Future

- **3.0 in development**
- **Support gfortran-15**
  - Including unsigned type
- **Allocatable arrays of derived types**
- **Unicode**
  - selected_char_kind('ISO_10646')
- **Significant speed-ups as well**

How can you help?

- I need Fortran examples
- Many combinations of even simple things need test cases
- Try it on your code and if it doesn’t work open a bug report.

Summary

- Gfort2py is a simple and easy to use Fortran to Python interface layer
- Supports many newer Fortran features
- Minimises changes to Fortran code and build system
- https://github.com/rjfarmer/gfort2py