

Fortran: calidad, empaquetamiento e interfaz con otros lenguajes

Empaquetamiento

Contenido

- Compilación.
- Makefiles.
- fpm.

Compilación

Compilación: Pasos básicos

Fortran es un lenguaje compilado.

Compilación básica

```
gfortran source.f90 -o main
```

Compilación: Flags importantes

```
-Wall -Wextra           ; Muchas warnings
-fcheck=all             ; checks durante runtime
-ON (N=0, 1, 2, 3, fast); grado de optimización
-funroll-loops          ; "desarmar" los loops
-fimplicit-none         ; forzar implicit none
-fdefault-real-8        ;
-freal-4-real-8         ;
```

<https://gcc.gnu.org/onlinedocs/gfortran/Fortran-Dialect-Options.html>

<https://gcc.gnu.org/onlinedocs/gfortran/Code-Gen-Options.html>

Makefiles.

Una forma de simplificar el proceso de compilación es mediante un Makefile.

- Forma estructurada de definir dependencias.
- Automatización de comandos.
- Cambiar el proceso de compilación con variables.

Makefiles: Ejemplo básico

■ Ejemplo: makefiles dir

Algo mejor:

<https://aoterodelaroza.github.io/devnotes/modern-fortran-makefiles/>

fpm

fpm: Fortran Package Manager

Gestor de paquetes desarrollado por fortran-lang, basado en el gestor cargo de Rust.

<https://fpm.fortran-lang.org/>

fpm: Funcionamiento

Command-line features

- handles (sub)module interdependencies
- automatically finds executables and tests
- runs executables and examples
- wildcard globbing for selection
- finds and runs test executables
- allows running tests in debugger
- automatically installs project
- user prefix used by default
- updates all dependencies
- basic caching and locking
- creates new projects with fpm layout
- backfilling to update existing projects



Fuente: <https://tcevents.chem.uzh.ch/event/14/contributions/68/>

fpm: Estructura de archivos

Layout of an fpm project

```
./
├── fpm.toml ← # Package manifest
├── src/
│   ├── lib_api.f90
│   ├── lib_version.f90
│   └── ...
├── app/
│   ├── cli.f90
│   ├── main.f90
│   └── ...
├── example/
│   ├── example_callback.f90
│   ├── example_highlevel.f90
│   └── ...
└── test/
    ├── main.f90
    ├── test_api.f90
    └── ...
```

```
# Package manifest
name = "lib"

[[library]]
source-dir = "src"

[[executable]]
name = "runner"

[build]
link = "nlopt"
```

- package manifest in **TOML**
- limited complexity of build file
- mainly meta data of project (license, author, keywords, ...)
- default layout requires only **name** (**src**, **app**, **example**, and **test**)
- **source-dir** for customizing layout
- executable names from **program** unit
- system libraries can be linked

fpm: Configuración

Un paquete de Fortran distribuido mediante fpm se configura mediante un archivo

`fpm.toml`

Contiene

- Metadata del proyecto
- Dependencias (separando dependencias de desarrollo de release)
- Librerías de sistema que se utilizan

```
name = "feos"
version = "0.1.0"
license = "MIT"
author = "Federico Benelli"
maintainer = "federico.benelli@mi.unc.edu.ar"
copyright = "Copyright 2022, Federico Benelli"

[build]
@link = ["lapack", "blas"]
auto-executables = true
auto-tests = true
auto-examples = true

[[executable]]
name = "ex"
source-dir = "app"
main = "main.f90"

[install]
library = true

[dependencies]
ftools.git = "https://github.com/fedebenelli/ftools"
nlopt-f.git = "https://github.com/grimme-lab/nlopt-f"

[dev-dependencies]
# Testing dependencies
test-drive.git = "https://github.com/fortran-lang/test-drive"
```

Figure 1: `fpm.toml`

Calidad

Análisis estático

Análisis estático: Linters

- fprettify.
- flinter.
- fortran-linter.

Un set de guidelines de estilo:

https://flinter.readthedocs.io/en/latest/fortran_guidelines.html

Linters: fprettify

- No es un linter, es un formateador
- Solo hace tres cosas (que es más o menos lo mismo)
 - Alinea indentaciones.
 - Acomoda whitespace entre operadores.
 - Trata de limitar el ancho de filas. Tira un warning si no puede

`fprettify --whitespace 4 -l 80 file.f90`

```
program main
!comment
implicit none
real*8 x
real*4 y(3)
integer i

x=2.3
y =[2, 3, 1]

do i=1, 3
  y( i) = 2*i ** 2
end do

end program main
```

```
program main
!comment
implicit none
real * 8 x
real * 4 y(3)
integer i

x = 2.3
y = [2, 3, 1]

do i = 1, 3
  y(i) = 2 * i**2
end do

end program main
```

Linters: fortran-linter

Este si es un linter.

- Sigue siendo más limitado que flake8
- Tira un set de warnings y también tiene la posibilidad de modificar el código

```
# Mostrar que tan mal estamos  
fortran-linter --syntax-only --line-length 80 file.f90  
# Modificar el archivo  
fortran-linter --line-length 80 --inplace file.f90
```

Si se modifica el archivo se genera un backup file.f90.orig

Linters: flinter

Por último, también está flinter

- Es un poco más avanzado.
- Tiene un archivo de configuración para las regex.
- Le da un puntaje al código.

```
pip install flinter
```

```
flint lint file.f90
```

```
flint lint folder
```

```
flint tree folder #> Está bueno para saber donde atacar con
```

Testing

Testing

Existen varios frameworks de testing:

- FUnit
- veggies
- test-drive

Yo uso test-drive así que vamos a ver ese.

Testing: test-drive

Los tests se separan en módulos que luego se colectan desde un runner principal.

- Puede agregarse como una dependencia en fpm.
- Es un único archivo que también puede agregarse a la fuente.

test-drive: test

```
subroutine test_cosa(error)
  type(error_type), allocatable, intent(out) :: error

  call check(error, 1 + 2 == 3)
  if (allocated(error)) return

  ! equivalent to the above
  call check(error, 1 + 2, 3)
  if (allocated(error)) return
end subroutine test_cosa
```

test-drive: collector de tests

```
!> Collect all exported unit tests
subroutine collect_suite(testsuite)
  !> Collection of tests
  type(unittest_type), allocatable, intent(out) :: testsuite(:)

  testsuite = [ &
    new_unittest("valid", test_valid), &
    new_unittest("invalid", test_invalid, should_fail=.true.) &
  ]

end subroutine collect_suite
```


test-drive: runner

```
program tester
...
use test_suite, only : collect_suite
...
type(testsuite_type), allocatable :: testsuites(:)

stat = 0

testsuites = [ &
  new_testsuite("suite1", collect_suite), &
  ...
]

do is = 1, size(testsuites)
  write(error_unit, fmt) "Testing:", testsuites(is)%name
  call run_testsuite(testsuites(is)%collect, error_unit, stat)
end do

if (stat > 0) then
  write(error_unit, '(i0, 1x, a)') stat, "test(s) failed!"
  error stop
end if

end program tester
```

Coverage

Coverage

Se pueden generar archivos de coverage compilando con la flag `--coverage` (usando `gfortran`).

- Con `fpm` simplemente es necesario correr:
`fpm test --flag "--coverage"`
- `gcovr`: Es un paquete en python que warpea la herramienta `gcov` y brinda una funcionalidad similar a `coverage.py`
 - Importante aclararle de donde no incluir coverage.

```
gcovr --exclude "build" \  
      --exclude "example" \  
      --exclude "app" --fail-under-branch 90
```

Documentación

Documentación: Ford

La generación de documentación se puede hacer mediante Ford

Genera automáticamente un html a partir del código.

También permite agregar páginas estáticas, ya sean html o markdown.

Ford: Archivo de configuración

Ford se maneja con un único archivo de configuración donde se especifican los detalles de qué documentar y cómo.

- Ejemplo: feos

Otras cosas

Otras cosas

Existen varias herramientas para mejorar Fortran en mal estado. No se puede llegar a evaluarlas dentro del scope del curso (ni yo llegué a usarlas), pero dejo el link a una compilación por si es de utilidad para alguien:

- Roquefort: transforma bloques `common` a módulos y solo importa variables/rutinas que se usan
 - <https://github.com/NLESC-JCER/roquefort>
- f90wrap: Genera wrappers para objetos que no son compatibles con f2py (lo que sigue)
 - <https://github.com/jameskermode/f90wrap>

Más herramientas:

- <https://github.com/Beliavsky/Fortran-Tools#refactoring>

Interfaces con otros lenguajes

C

C

A partir de Fortran 2003 se puede correr código Fortran desde C (y viceversa)

Fortran

```
!function.f09  
subroutine f(x, y) bind(C, name="f_fortran")  
  use iso_c_binding, only: c_double  
  real(c_double), intent(in) :: x  
  real(c_double), intent(out) :: y  
  y = x**2  
end subroutine f
```

C

```
// cside.c  
#include <stdio.h>  
extern double f_fortran(double [], double []);  
main()  
{  
  double x=2.0; double y=5.0; double z;  
  z = f_fortran(&x, &y);  
  printf("x: %f, y: %f, z: %f",  
        x, y, z);  
}
```

Compilación

```
gfortran -c function.f90  
gcc -o exec cside.c function.f90
```

Python

El principal método de interfacear Fortran con Python es mediante f2py

- Genera módulos de extensión (extension modules) para Python a partir de código Fortran.
 - Utilizar subrutinas, datos en bloques COMMON y variables en módulos de FORTRAN 77 o Fortran 90/95 desde Python.
 - Llamar funciones de Python desde Fortran (callbacks).
 - Manejar automáticamente la diferencia entre arrays NumPy-contiguos (esto es, C-contiguos) y Fortran-contiguos.
- Fue creado en 1999 por Pearu Peterson mientras era estudiante de doctorado en la Universidad Técnica de Tallin, y en 2005 después de varias versiones estables quedó incluido dentro de NumPy.

f2py: Instalación

- Viene instalado con numpy, así que no hace falta nada
- Hace falta tener instalado un compilador de fortran (preferentemente gfortran)

Uso

Se puede usar desde la terminal como:

```
f2py -c -m modulo_python archivo_fortran.f90
```

f2py: Lo malo

Incompatibilidad

f2py está limitado a las características básicas de Fortran. Objetos (types) no funcionan.

```
subroutine sub_noanda(x, y)
  type(mi_objeto) :: x
  real :: y
  !cosas
end subroutine
```

Solución

Hacer subrutinas/funciones wrappers que tomen la información básica

f2py: lo malo

```
subroutine sub_noanda(x, y)
  type(mi_objeto), intent(in) :: x
  real, intent(out) :: y
  !cosas
end subroutine

subroutine sub_estasi(n, x, y)
  integer :: n
  real, intent(in) :: x(n)
  real, intent(out) :: y

  type(mi_objeto) :: in_obj

  in_obj%x = x
  call sub_noanda(in_obj, y)
end subroutine
```

Si el código ya es extremadamente complicado lo mejor es armar un par de procedimientos que llamen a lo importante y solo se trabaje con eso, compilando previamente el código principal como una librería.

f2py: Packaging

Muy lindo compilarlo desde la terminal, pero ¿para automatizarlo en un paquete?

Es necesario utilizar CMake.

- Ejemplo gchop.
- Ejemplo pff.