



SKILLPILLS

Skill Pill: Julia

Lecture 1: Introduction

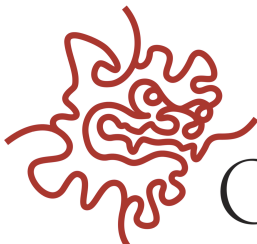
James Schloss Valentin Churavy

Okinawa Institute of Science and Technology

james.schloss@oist.jp

valentin.churavy@oist.jp

July 7, 2017



OIST

- 1 The foreign world, using Julia to reuse prior work
- 2 Macros and metaprogramming
- 3 The Julia compiler
- 4 Performance
- 5 Performance analysis

Julia allows you to use other languages (such as Fortran or C) by using the `ccall` function:

```
julia> t = ccall(:clock, "libc"), Int32, ())  
2292761
```

Here, we are calling the `clock` function from the `libc` library in C.

Let's say you want to use a simply multiply function in Fortran:

```
!! We'll be using subroutines instead of functions
subroutine multiply(A, B, C)
    REAL*8 :: A, B, C
    C = A * B
    return
end
```

or C:

```
// Nothing fancy here...
double multiply(double A, double B){
    return A*B;
}
```

In order to use your favorite C or Fortran code in Julia, you need to compile it into a library, like so:

```
gcc -shared -O2 multiply.c -fPIC -o c_multiply.so  
gfortran -shared -O2 multiply.f90 -fPIC -o  
    fortran_multiply.so
```

In order to use your favorite C or Fortran code in Julia, you need to compile it into a library, like so:

```
gcc -shared -O2 multiply.c -fPIC -o c_multiply.so  
gfortran -shared -O2 multiply.f90 -fPIC -o  
    fortran_multiply.so
```

These will create libraries with all of the necessary functions you could want, but beware:

C and Fortran compilers mangle function names!

There are 3 things to keep. Make sure you

- 1 Have the right mangled name
- 2 Are using the right type
- 3 Are using the function correctly.

There are 3 things to keep. Make sure you

- 1 Have the right mangled name
- 2 Are using the right type
- 3 Are using the function correctly.

For example, in C:

```
# This function multiplies a and b into c by using the
# created C library
function call_c()
    a = Cdouble(1.0)
    b = Cdouble(3.0)
    c = ccall(:multiply, "/full/path/to/c_multiply"),
        Cdouble, (Cdouble, Cdouble), a, b)
    println(c)
end
```

Pointers are okay! For example, in Fortran:

```
# This function multiplies a and b into c by using
# the created FORTRAN library
function call_fortran()
    a = Cdouble[1.0]
    b = Cdouble[2.0]
    c = Cdouble[0.0]
    ccall((:multiply_, "/full/path/to/fortran_multiply"),
        Void, (Ptr{Float64}, Ptr{Float64}, Ptr{Float64}),
            a, b, c)
    println(c[1])
end
```

Pointers are okay! For example, in Fortran:

```
# This function multiplies a and b into c by using
# the created FORTRAN library
function call_fortran()
    a = Cdouble[1.0]
    b = Cdouble[2.0]
    c = Cdouble[0.0]
    ccall((:multiply_, "/full/path/to/fortran_multiply"),
        Void, (Ptr{Float64}, Ptr{Float64}, Ptr{Float64}),
            a, b, c)
    println(c[1])
end
```

More information can be found here: <https://docs.julialang.org/en/stable/manual/calling-c-and-fortran-code/>

Python <https://github.com/JuliaPy/PyCall.jl>

R <https://github.com/JuliaInterop/RCall.jl>

C++ <https://github.com/Keno/Cxx.jl>

Matlab I have heard rumours of such a thing existing, but the horror
...

Conclusion

Start writing Julia code now without being worried about losing your prior work!

- 1 Surface syntax (the code you write)
- 2 Desugared AST `@code_lowered`
- 3 Type-inferred AST `@code_typed`
- 4 LLVM IR `@code_llvm`
- 5 Native assembly `@code_native`

Measure first

Before you start iterating on your code establish a baseline performance. Computers are noisy system so we use the lowest runtime as a metric.

Measure first

Before you start iterating on your code establish a baseline performance. Computers are noisy system so we use the lowest runtime as a metric.

- Check for type-instabilities with `@code_warntype`
- Measure runtime and allocations with `@time`
- Benchmark using `@btime`, and `@benchmark` from `BenchmarkTools.jl`
- Profiler and `ProfileView.jl`
- Memory Allocation tracker

Measure first

Before you start iterating on your code establish a baseline performance. Computers are noisy system so we use the lowest runtime as a metric.

- Check for type-instabilities with `@code_warntype`
- Measure runtime and allocations with `@time`
- Benchmark using `@btime`, and `@benchmark` from `BenchmarkTools.jl`
- Profiler and `ProfileView.jl`
- Memory Allocation tracker

Read the performance tips section of the Julia manual <https://docs.julialang.org/en/stable/manual/performance-tips/>

You can profile a piece of code with Julia's inbuilt profiler.

- Profile a specific function

- Clear the recorded profile

- Print the profile

- Print the profile including stacktraces reaching into C.

ProfileView.jl

The textual output of the profiler can be hard to understand. ProfileView.jl gives a graphical representation.

```
using ProfileView
ProfileView.view()
```

To track memory allocations you have to start Julia with the memory allocation tracker enabled.

```
# Track only allocation in user code
julia --track-allocation=user
# Track allocation in all code (includeing the Julia base)
julia --track-allocation=all
```

After quitting Julia *.mem files are created that contain cumulative amounts of allocated memory.

Getting useful data

Since we have to start Julia with track allocations enable we will gather a lot of noisy data. To cut down the noise run your code in a session once and then use `Profile.clear_malloc_data()` to reset the allocation counts and then run your code again only tracking revelant allocations.

```
function mysum(A)
    acc = 0
    for x in A
        acc += x
    end
    return acc
end
```

```
function myfun()
    s = 0.0
    N = 10000
    for i=1:N
        s+=det(randn(3,3))
    end
    s/N
end
```

`e.print()` file `NextSession` Data Structures and Algorithms

`Last Session` Parallel computing, threading, GPUs? Up to grabs.