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

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.

Your legacy code

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

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

or C:

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

Preparing your legacy code

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
```

Preparing your legacy code

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!

Using your legacy code

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.

Using your legacy code

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
```

Using your legacy code

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
```

Using your legacy code

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/`

Support for other languages

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!

The stages of the compiler

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`

Macros and metaprogramming I

Julia has powerful metaprogramming facilities that allow you to manipulate the AST (abstract syntax tree) of a Julia program.

```
prog = "1 + 1"
ex1 = parse(prog) # Parse a Julia program and return the
    expression tree
# This is equal to manually constructing the expr tree
ex2 = Expr(:call, :+, 1, 1)
# Or by using quote
ex3 = :( 1 + 1)
dump(ex1) # Inspect an expr tree
```

The key point here is that Julia code is internally represented as a data structure that is accessible from the language itself.

```
# use eval to execute an expression
eval(ex1)
```

Macros

`eval` works at runtime and you should avoid it as much as possible. You can use macros to do syntax transformations statically. Macros always start with an `@`.

```
# simplistic implementation of the @assert macro in Julia
macro assert(ex)
    return :( $ex ? nothing :
        throw(AssertionError($(string(ex)))) )
end

@assert 1 == 1.0
@assert 1 == 0

@macroexpand @assert 1 == 0
```

Methodology

Methodology

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

Methodology

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/`

About global scope

Performance tips

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible.
Any code that is performance critical or being benchmarked should be inside a function.

About global scope

Performance tips

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible.
Any code that is performance critical or being benchmarked should be inside a function.

The natural unit of compilation is a function. Code in global scope does not get compiled.

Type instabilities

Julia relies on type-inference to generate optimal code, when your code has type-instabilities your code will execute slower. Some of these problems can be fixed by compiler improvements over time, but if you want the fast code now they are best avoided, especially in hot code. Sometimes introduction a function barrier can help generating optimal code.

Type instabilities

Julia relies on type-inference to generate optimal code, when your code has type-instabilities your code will execute slower. Some of these problems can be fixed by compiler improvements over time, but if you want the fast code now they are best avoided, especially in hot code. Sometimes introduction a function barrier can help generating optimal code.

```
typeinstable() = rand() < 0.5 ? 1.0 : 1
```

Using BenchmarksTools.jl

BenchmarkTools provides a variety of tools to run benchmark and compare them. See `https://github.com/JuliaCI/BenchmarkTools.jl`

```
julia> using BenchmarkTools
julia> @benchmark sin(1)
julia> @benchmark sum(rand(1000))
# Interpolate inputs and globals into your benchmark to measure
    the right thing.
julia> @benchmark sum(\$(rand(1000)))
```

Using the Profiler

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

`@profile fun()`
: Profile a specific function

`Profile.clear()`
: Clear the recorded profile

`Profile.print()`
: Print the profile

`Profile.print(C=true)`
: Print the profile including stacktraces reaching into C.

ProfileView.jl

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

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

Using the memory allocation tracker

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 quiting 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.

A simple example

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

A supposedly simple task

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

What is next?

Next Session Data Structures and Algorithms

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