


UNIVERSITY OF L'AQUILA AND INFN



Introduction to the Julia Programming Language

Mosè Giordano ( @giordano)

Astro Hack Week 2017 @ University of Washington, Seattle

September 1, 2017

Why Julia?

The Julia Manifesto

In short, because **we are greedy**.

[...]

We want a language that's **open source**, with a liberal license. We want the **speed** of C with the **dynamism** of Ruby. We want a language that's **homoiconic**, with **true macros** like Lisp, but with obvious, familiar **mathematical notation** like Matlab. We want something as usable for **general programming** as Python, as **easy for statistics** as R, as natural for **string processing** as Perl, as powerful for **linear algebra** as Matlab, as good at **gluing programs together** as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it **interactive** and we want it **compiled**.

(Did we mention it should be as fast as C?)

<https://julialang.org/blog/2012/02/why-we-created-julia>

Ousterhout's Dichotomy

Programming languages tend to fall into one of the two following groups:

System programming languages (or applications languages)

- statically typed
- programs compiled
- support creating complex data structures
- usually fast
- hard to code and do it efficiently

Scripting languages (or glue languages)

- dynamically typed
- scripts interpreted
- little or no provision for complex data structures
- usually slow
- easy to code

Ousterhout's Dichotomy

Programming languages tend to fall into one of the two following groups:

System programming languages
(or applications languages)

- statically typed
- programs compiled
- support creating complex data structures
- usually fast
- hard to code and do it efficiently

Scripting languages
(or glue languages)

- dynamically typed
- scripts interpreted
- little or no provision for complex data structures
- usually slow
- easy to code

Solving the Two-Language Problem



- dynamically typed
- JIT-compiled scripts
- users can easily define complex custom types, as efficient as standard ones
- easy to code

Solving the Two-Language Problem



- dynamically typed
- JIT-compiled scripts
- users can easily define complex custom types, as efficient as standard ones
- easy to code
- bonus feature: fast

Julia's Facts

- Latest stable version: **v0.6.0** (2017-06-19)
- **v1.0.0** expected by 2018 (hopefully!)
- More than **1 million downloads**
- Development started in 2009 at **MIT**, first public release in 2012
- Julia **users, partners** and **employers hiring Julia programmers** in 2017 include Amazon, Apple, BlackRock, Capital One, Comcast, Disney, Facebook, Ford, FRBNY, Google, Grindr, IBM, Intel, KPMG, Microsoft, NASA, Oracle, PwC, Raytheon and Uber
- Julia adoption is growing rapidly in **finance, energy, robotics, genomics**
- It is now being used and taught in several **universities** (<https://julialang.org/teaching/>)



Main Julia's Features

- Multiple dispatch
- Dynamic type system
- Good performance, approaching that of statically-compiled languages
- Built-in package manager
- Lisp-like macros and other metaprogramming facilities
- Syntactic loop fusion
- Call Python functions: use the `PyCall` package
- Call C and Fortran functions directly: no wrappers or special APIs
- Powerful shell-like capabilities for managing other processes
- Designed for parallelism and distributed computation
- User-defined types are as fast and compact as built-ins
- Automatic generation of efficient, specialized code for different argument types
- Elegant and extensible conversions and promotions for numeric types
- Efficient support for Unicode, including but not limited to UTF-8
- MIT licensed: free and open source

The Type System

Types can be

- **abstract** (e.g., `Number`, `AbstractArray`, `AbstractFloat`)
- **concrete** (e.g., `Int16`, `Vector`, `Float64`)
 - **primitive**: concrete type whose data consists of plain old bits (e.g., `Char`, `UInt8`, `Float32`)
 - **composite**: collection of named fields, an instance of which can be treated as a single value. They are called records, structs, or objects in various languages
 - **immutable** (e.g., `Complex`, `StepRangeLen`)
 - **mutable** (e.g., `BigFloat`, `String`)

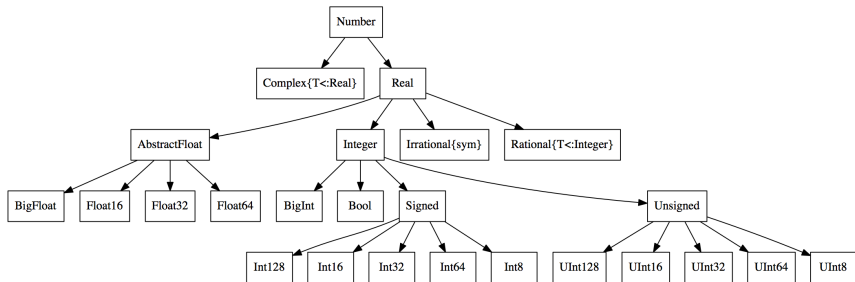
Parametric Types

Types can take **parameters**, so that type declarations actually introduce a **whole family of new types** — one for each possible combination of parameter values.

Because Julia is a **dynamically typed** language and doesn't need to make all type decisions at compile time, many traditional difficulties encountered in static parametric type systems can be relatively easily handled.

```
# `T` is the type of the elements,  
# `N` is the number of dimensions of the array  
mutable struct Array{T,N} <: DenseArray{T,N} end  
  
# `Vector` and `Matrix` are particular case of arrays  
const Vector{T} = Array{T,1}  
const Matrix{T} = Array{T,2}
```

Type Hierarchy of Numbers



Credits: Cormullion

Multiple Dispatch

- The same function identifier can have several **methods**
- Each method is specialized on the **number of arguments** and their **type**
- A **method “belongs” to all of its arguments**, not only to the first one
- It is possible to define different methods **specialized on different types**:
 - A method for integers
 - a method for rational numbers
 - a method for irrational numbers
 - a method for floating point numbers
 - etc...
- Dispatch is **dynamic on all arguments** (in C++/Java: dynamic on first argument, static for the others)
- Multiple dispatch **saves you from branching** (**if ... else ... end**)

```
julia> methods(+)  
# 180 methods for generic function "+":  
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:260  
[2] +(x::Bool, y::Bool) in Base at bool.jl:92  
...  
[180] +(a, b, c, xs...) in Base at operators.jl:467
```

Multiple Dispatch: An Example

Define the types

```
# The abstract type `Item`  
abstract type Item end  
# Followings are subtypes of the abstract type `Item`  
struct Paper    <: Item end  
struct Rock     <: Item end  
struct Scissors <: Item end
```

Define the rules of the game

```
play(::Type{Paper}, ::Type{Rock})      = "Paper"  
play(::Type{Paper}, ::Type{Scissors}) = "Scissors"  
play(::Type{Rock},   ::Type{Scissors}) = "Rock"  
play(a::Type{T},    b::Type{T}) where {T<:Item} =  
    "Draw, try again"  
play(a, b) = play(b, a) # Commutativity
```

Multiple Dispatch: An Example (cont.)

Let's play!

```
julia> play(Scissors, Rock)
```

```
"Rock"
```

```
julia> play(Scissors, Scissors)
```

```
"Draw, try again"
```

```
julia> play(Rock, Paper)
```

```
"Paper"
```

```
julia> play(Scissors, Paper)
```

```
"Scissors"
```

Multiple Dispatch: An Example (cont.)

Add some randomness

```
julia> rand()  
0.459521156680214
```

```
julia> rand([2, 3, 5, 7])  
5
```

```
julia> subtypes(Item)  
3-element Array{Union{DataType, UnionAll},1}:  
 Paper  
  Rock  
 Scissors
```

```
julia> rand(subtypes(Item))  
Rock
```

```
julia> play(rand(subtypes(Item)), rand(subtypes(Item)))  
"Paper!"
```


How Is Multiple Dispatch Useful

Multiple dispatch is particularly useful for mathematical code, where it makes little sense to artificially deem the operations to “belong” to one argument more than any of the others: does the addition operation in $x + y$ belong to x any more than it does to y ? The implementation of a mathematical operator generally depends on the types of all of its arguments.

Read more at <https://docs.julialang.org/en/stable/manual/methods/>

How Is Multiple Dispatch Useful: An Example

Divisions involving complex numbers in Python:

```
>>> import numpy
>>> numpy.complex128(1) / 0
(inf+nan*j) # This is correct
>>> numpy.float64(1) / numpy.complex128(0)
(inf+nan*j) # Should be (nan+nan*j)
>>> numpy.complex128(1) / numpy.complex128(0)
(inf+nan*j) # Should be (nan+nan*j)
>>> numpy.float64(1) / numpy.complex128(1)
(1+0j)      # Should be (1-0*j)
```

Not compliant with IEEE standard :-(

How Is Multiple Dispatch Useful: An Example (cont.)

Divisions involving complex numbers in Julia:

```
julia> complex(1) / 0
Inf + NaN*im

julia> 1 / complex(0)
NaN + NaN*im

julia> complex(1) / complex(0)
NaN + NaN*im

julia> 1 / complex(1)
1.0 - 0.0im
```

Compliant with IEEE standard and consistent with mathematical definition of limit of complex numbers :-)

Metaprogramming

- Like Lisp, Julia is **homoiconic**: it represents **its own code as a data structure of the language** itself
- Since code is represented by objects that can be created and manipulated from within the language, it is possible for a program to **transform and generate its own code**. This allows sophisticated code generation without extra build steps, and also allows true Lisp-style macros operating at the level of abstract syntax trees (ASTs)
- In contrast, preprocessor "macro" systems, like that of C and C++, perform **textual manipulation and substitution** before any actual parsing or interpretation occurs
- Julia's macros allow you to modify an **unevaluated expression** and return a new expression at **parsing-time**
- Macros allows the creation of **domain-specific languages** (DSLs; e.g., JuMP.jl for mathematical optimization, DifferentialEquations.jl for solving differential equations). See <https://julialang.org/blog/2017/08/dsl>

For more information, read the manual:

<https://docs.julialang.org/en/stable/manual/metaprogramming.html>

Macros: Example 1

Evaluating a polynomial with real coefficients

- The slow way:

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n$$

- The fast way (Horner's method):

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)))$$

This is fast, but hard to read and write

Stepping through an array of coefficients at **run-time** is slow. Macros gives you the desired expression at **parsing-time**, which will be quickly evaluated at run-time.

Macros: Example 1 (cont.)

```
julia> @evalpoly 3.1 9.4 2.7 0.6
```

```
23.536
```

```
julia> @macroexpand @evalpoly 3.1 9.4 2.7 0.6
```

```
:(let #88#tt = 3.1
    #= math.jl:139 =#
    if #88#tt isa Base.Math.Complex
        [...]
    else
        Base.Math.t = #88#tt
        (Base.Math.muladd)(Base.Math.t, (Base.Math.muladd)(
            Base.Math.t, 0.6, 2.7), 9.4)
    end
end)
```

Macros: Example 2

Definition of @time macro:

```
macro time(ex)
  quote
    local stats = gc_num()
    local elapsedtime = time_ns()
    local val = $(esc(ex))
    elapsedtime = time_ns() - elapsedtime
    local diff = GC_Diff(gc_num(), stats)
    time_print(elapsedtime, diff.allocd, diff.total_time,
              gc_alloc_count(diff))
  val
end
end
```

Macros: Example 2 (cont.)

Measuring running time of functions

```
julia> @time 6.4sin(0.5)
0.002793 seconds (532 allocations: 31.260 KiB)
3.0683234470668994

julia> @time 6.4sin(0.5)
0.000008 seconds (6 allocations: 192 bytes)
3.0683234470668994

julia> @time exp(4.8) * cos(6.3) / gamma(7.1)
0.028400 seconds (11.94 k allocations: 696.640 KiB)
0.13981504275638063

julia> @time exp(4.8) * cos(6.3) / gamma(7.1)
0.000019 seconds (9 allocations: 240 bytes)
0.13981504275638063
```

Note: Smarter methods to benchmark functions are available in `BenchmarkTools.jl` package

Macros: Example 3

```
julia> @which exp10(-5.4)
exp10(x::Float64) in Base.Math at math.jl:293
```

```
julia> @code_llvm 1 + 2
```

```
define i64 @"jlsys+_57925"(i64, i64) #0 !dbg !5 {
top:
    %2 = add i64 %1, %0
    ret i64 %2
}
```

```
julia> @code_llvm 3.1 * 9.4
```

```
define double @"jlsys_*_57825"(double, double) #0 !dbg !5 {
top:
    %2 = fmul double %0, %1
    ret double %2
}
```

Syntactic Loop Fusion

```
julia> f(x) = 3x^2 + 5x + 2
f (generic function with 1 method)

julia> A = f.([-1.4, 3.7, 8.5])
3-element Array{Float64,1}:
 0.88
61.57
261.25

julia> A .= A .^ 2 .- log.(A)
3-element Array{Float64,1}:
 0.902233
3786.74
68246.0

julia> using BenchmarkTools

julia> @btime @. $A = $A ^ 2 - log($A);
25.103 ns (0 allocations: 0 bytes)
```

Syntactic Loop Fusion (cont.)

Advantages

- Julia recognizes the “vectorized” nature of the operations at a syntactic level (before e.g. the type of `x` is known), and hence the loop fusion is a **syntactic guarantee**, not a compiler optimization that may or may not occur for carefully written code
- The caller is able to “**vectorize**” **any function**, without relying on the function author
- Nested dot calls are **fused into a single loop**
- This feature is completely **general**, works with any type and any function

Read more at <https://julialang.org/blog/2017/01/moredots>

Calling Other Languages

Do you have legacy code that you still want to be able to use? Don't worry!

```
julia> ccall(:exp, :libm), Cdouble, (Cdouble,), 1.57)
4.806648193775178

julia> path = ccall(:getenv, "libc"), Cstring, (Cstring,),
    "SHELL")
Cstring(@0x00007fff5fbffc45)

julia> unsafe_string(path)
"/bin/bash"
```

Calling Other Languages (cont.)

```
julia> using PyCall

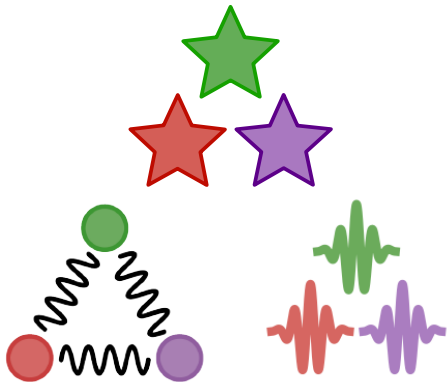
julia> @pyimport math

julia> math.sin(math.pi / 4) - sin(pi / 4)
0.0

julia> @pyimport numpy.random as nr

julia> nr.rand(3,4)
3×4 Array{Float64,2}:
 0.971683  0.408847  0.00718184  0.827758
 0.811887  0.568809  0.56686   0.637123
 0.120899  0.574567  0.218255  0.602889
```

If you come to Julia from another language, keep in mind the following differences: <https://docs.julialang.org/en/stable/manual/noteworthy-differences>



My Julia packages:

- `AstroLib.jl`
- `Cuba.jl`
- `Deconvolution.jl`
- `LombScargle.jl`
- `Measurements.jl`
- `PolynomialRoots.jl`

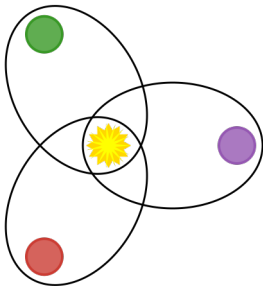
Find out more at

<https://github.com/giordano>

Advertisement: AstroLib.jl

Mostly a porting of routines from IDL
AstroLib with extra goodies

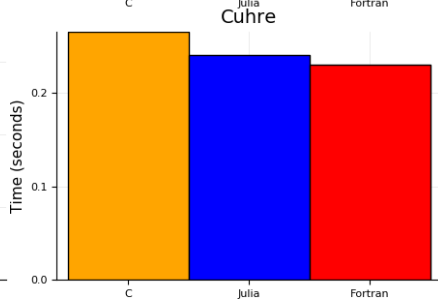
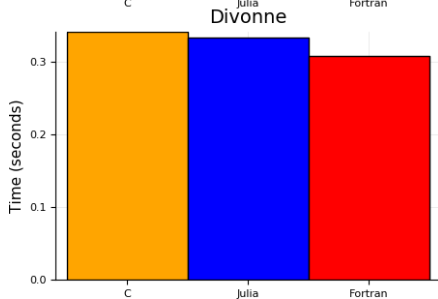
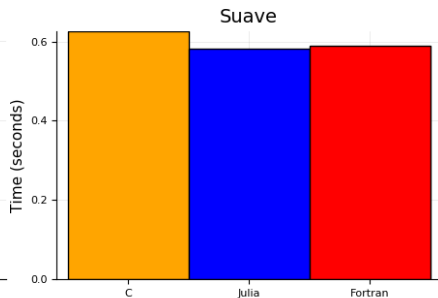
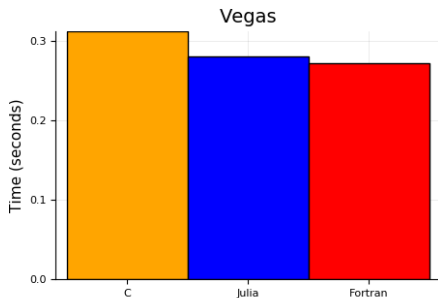
- Coordinates and positions
- Time and date
- Moon and sun
- Generic utilities



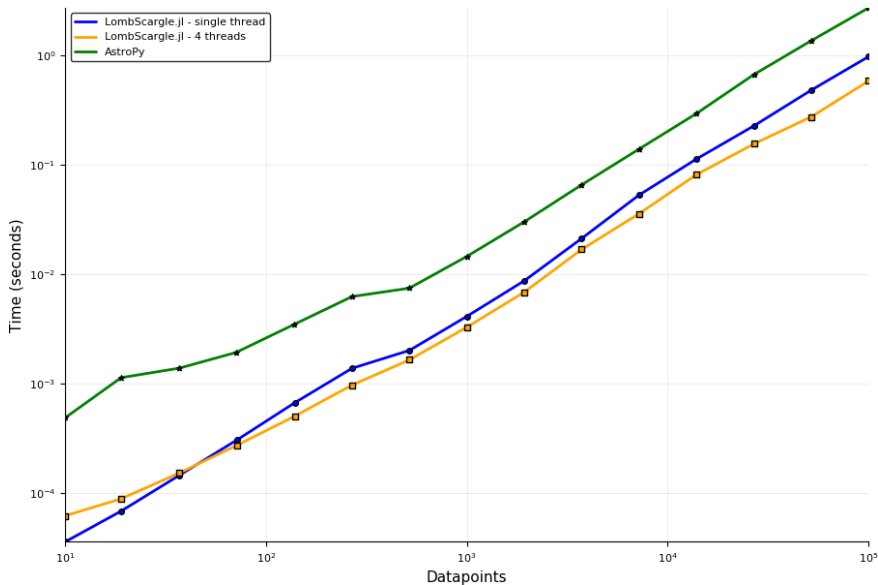
Google
Summer of Code

In 2017 became the first astronomical software written in Julia to be funded by Google Summer of Code

Advertisement: Cuba.jl



Advertisement: LombScargle.jl



Advertisement: Measurements.jl

```
julia> using Measurements # Load the package
```

```
julia> l = 0.936 ± 1e-3  
0.936 ± 0.001
```

```
julia> typeof(l)  
Measurements.Measurement{Float64}
```

```
julia> supertype(Measurement)  
AbstractFloat
```

```
julia> T = 1.942 ± 4e-3  
1.942 ± 0.004
```

```
julia> g = 4pi ^ 2 * l / T ^ 2  
9.797993213510699 ± 0.041697817535336676
```

```
julia> @uncertain zeta(1.4 ± 0.2)  
3.1055472779775815 ± 1.236240133830008
```

Advertisement: Measurements.jl (cont.)

```
julia> x = 8.4 ± 0.7 # An independent measurement  
8.4 ± 0.7
```

```
julia> u = 2x # Functionally correlated with x  
16.8 ± 1.4
```

```
julia> (x + x) - u  
0.0 ± 0.0
```

```
julia> u / 2x  
1.0 ± 0.0
```

```
julia> u^3 - (2x^3 + 6x * x ^ 2)  
0.0 ± 0.0
```

```
julia> cos(x)^2 - (1 + cos(u))/2  
0.0 ± 0.0
```

Advertisement: Measurements.jl (cont.)

```
julia> beta(x, u) - gamma(x)*gamma(u)/gamma(3x)
0.0 ± 1.852884572118782e-23

julia> cis(x) - exp(im * x)
(0.0 ± 0.0) + (0.0 ± 0.0)im

julia> A = [(14 ± 0.1) (23 ± 0.2); (-12 ± 0.3) (24 ± 0.4)]
2×2 Array{Measurements.Measurement{Float64},2}:
 14.0±0.1  23.0±0.2
-12.0±0.3  24.0±0.4

julia> inv(A) * A ≈ eye(A)
true

julia> QuadGK.quadgk(sin, -u/2, x)
(2.0122792321330962e-16 ± 0.0, 0.0)
```

Arbitrary precision, complex numbers and linear algebra operations work out-of-the-box, thanks to Julia's type system and its genericity

JuliaAstro Organization

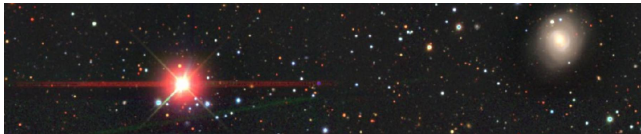
Packages in JuliaAstro

- AstroLib.jl
- AstronomicalTime.jl (WIP)
- Cosmology.jl
- DustExtinction.jl
- EarthOrientation.jl
- ERFA.jl
- FITSIO.jl
- WCS.jl

Other packages

- CasaCore.jl
- JPLEphemeris
- LibHealpix.jl
- OIFITS.jl
- SkyCoords.jl





Project goals:

- ❶ Catalog all galaxies and stars that are visible through the next generation of telescopes
 - The Large Synoptic Survey Telescope (LSST) will house a 3200-megapixel camera producing 15 TB of images nightly
- ❷ Replace non-statistical approaches to building astronomical catalogs from photometrical data
- ❸ Identify promising galaxies for spectrograph targeting
 - Better understand dark energy and the geometry of the Universe
- ❹ Develop and extensible model and inference procedure, for use by the astronomical community
 - Future applications might include finding supernovae and detecting near-Earth asteroids

Accomplishments:

- ❶ Reached 1.54 petaFLOPS performance (first First Julia application to exceed 1 petaFLOPS)
 - Julia is probably the first dynamic high-level language to enter the petaFLOPS club (other languages in it: Assembly, Fortran, C/C++)
 - Code ran on 9300 Cori Phase II nodes
 - 1.3 million threads on 650 000 KNL cores
- ❷ Processed most of SDSS dataset in 14.6 minutes
 - Loaded and analyzed 178 TB
 - Optimized 188 million stars and galaxies
- ❸ First comprehensive catalog of visible objects with state-of-the-art point and uncertainty estimates
- ❹ Demonstration of Variational Inference on 8 billion parameters
 - 2 orders of magnitude larger than other reported results

Discover more at <https://www.youtube.com/watch?v=uecdcADM3hY>

Take-Home Messages

- Julia is a **modern, high-level, high-performance dynamic** programming language for numerical computing
- Designed for **parallelism** and **distributed computation**
- Smart **type system**
- Julia programs are organized around **multiple dispatch**
- **Metaprogramming** capabilities
- Most of **Julia is written in Julia** itself
- Caveat: good performances are achieved by writing the code with some care (e.g., **type-stability**, see more at <https://docs.julialang.org/en/stable/manual/performance-tips.html>)
- My 2 cents: main Julia's strength is **genericity**, which increase **productivity**

Got Interested?

- Official website: <https://julialang.org/>
- Install Julia: <https://julialang.org/downloads/>
- Try Julia from the Browser: <https://juliabox.com/>
- Manual: <https://docs.julialang.org/en/stable/>
- Learning resources: <https://julialang.org/learning/>
- List of registered packages: <https://pkg.julialang.org/>
- GitHub repository: <https://github.com/JuliaLang/julia>
- Discussion forum: <https://discourse.julialang.org/>
- Plotting in Julia:
 - Matplotlib: <https://github.com/JuliaPy/PyPlot.jl>
 - Plots: <https://juliaplots.github.io/>
- MATLAB–Python–Julia cheatsheet:
<https://cheatsheets.quantecon.org/>