Introduction to Julia
It is all about the :Dots...

Stanislav Kazmin[1]

[1]Max Planck Institute for Mathematics in the Sciences

May 25, 2018

Content

Introduction

Julia is a high-level, high-performance dynamic programming language for numerical computing.[1]

- Download: `https://julialang.org/downloads/`
- Documentation: `https://docs.julialang.org`
- Products based on Julia: `https://juliacomputing.com/`
- JuliaBox (jupyter online): `https://www.juliabox.com/`

[1] *The Julia Language*, `https://julialang.org/`

Introduction

- current stable release: 0.6.2
- version 1.0 planned for 2018
- over 1.8 Million Downloads by January 2018
- used in companies like: Amazon, Apple, Disney, Facebook, Ford, Google, Grindr, IBM, Microsoft, NASA, Oracle and Uber[2]

[2] S. D. D'Cunha, *How A New Programming Language Created By Four Scientists Now Used By The World's Biggest Companies*, `https://www.forbes.com/sites/suparnadutt/2017/09/20/this-startup-created-a-new-programming-language-now-used-by-the-worlds-biggest-companies/`

Features

Julia Website

- good performance approaching C/C++
- macros and metaprogramming
- designed for parallelism and distributed computation
- user-defined types are as fast and compact as built-ins
- efficient support for Unicode
- MIT licensed: free and open source

Personal Favorites

- Julia is sweet (a lot of syntactic sugar)
- math-like notation
- fast by design
- easy to use but with high optional control
- tabs are not part of the syntax

Reasons to Use

Julia is general propose language with numerical computing in mind

Consider Julia if you[3]

- write your own algorithms
- want to prototype and scale up and analyze using the same language
- need easy access to parallel computing
- want your code be generic but fast
- prefer math-like short syntax and Unicode
- want to use automatic differentiation

Some Areas

- Data Research
- Machine Learning
- Statistics
- Financial Mathematics
- ...

[3] *John Pearson | Introduction to Julia for Pythonistas,*

History

Why We Created Julia?
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?)[4]

[4] *Why We Created Julia*, `https://julialang.org/blog/2012/02/why-we-created-julia`

History

- started in 2009 by Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman
- first public release 0.1 in 2012

Goals

Solve the "Two Language Problem"

- easy to learn and flexible
- fast and optimized

Improve the "Expression Problem"[5]

- add **new types** and use **implemented methods** versus add **new methods** for **implemented types** without the need of recompilation

[5] *Expression problem*, in *Wikipedia* (Jan. 28, 2018), `https://en.wikipedia.org/w/index.php?title=Expression_problem&oldid=822799993`

History

Inspiration

- C/C++: speed
- Fortran: Type names, indexing, column majority
- Lisp: metaprogramming, functional programming
- MATLAB: mathematical notation
- Python: simple control flow, API
- R: statistics

Design

on first call each function gets compiled before it runs

Just in Time Compiler (simplified)

$$\text{Julia code} \xrightarrow{\textit{parser}} \text{AST} \xrightarrow{\textit{LLVM}} \text{machine code}$$

Core Design Decision[6]

type-stability through specialization via **multiple-dispatch**

- design should be optimizable through LLVM
- Julia is (mainly) written in Julia itself through metaprogramming
- partly in C, assembly and its parser in Scheme (”FemtoLisp”)
- functions are first class elements and do **not** belong to a class

[6] *WhyJulia*, http://ucidatascienceinitiative.github.io/IntroToJulia/Html/WhyJulia

Types

Type of a Value

describes how the value is structured (stored in memory)

- compiler needs to know the type for optimization
- only types with exact known memory layout can be used fast
- other types are stored in the heap

Problem with Dynamical Languages

types are generally not known at compile time

Solution

multiple-dispatch and type-stability

Types in Julia

Type Stability

types of the parameters **exactly** define the type of the result

- in Julia **all** types are **generic** and defined in Julia itself[7]
- custom types are as fast as predefined types like `Float64`
- Julia uses a type hierarchy to define dependencies of types
- the types tree is fully connected with `Any` as top element
- **concrete types** (real instances in memory) are always leafs and cannot have sub-types → compiler knows exactly how the memory is structured
- **abstract types** only form the hierarchical structure of the tree

[7] *Types · The Julia Language*, https://docs.julialang.org/en/stable/manual/types/

Types in Julia

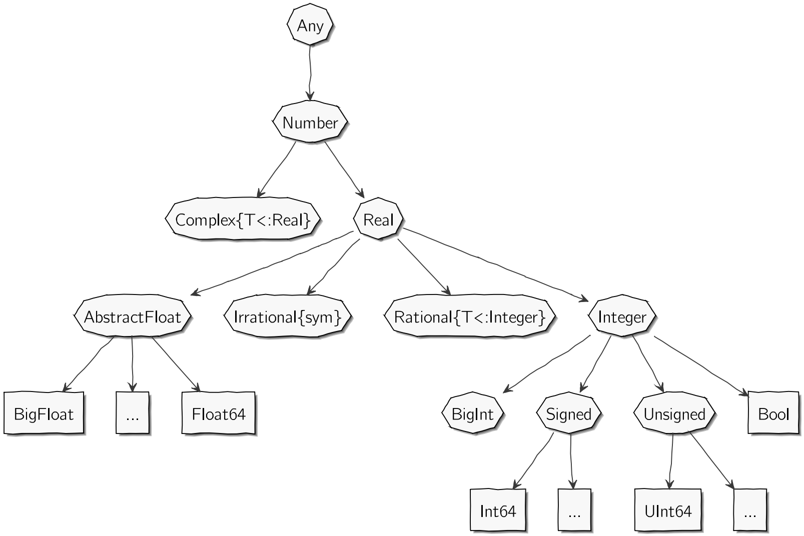

Figure 1: Number type hierarchy in Julia

Types in Julia

- **parametric types** are invariant

```julia
struct Point{T} # point with two coords of type T
  x::T
  y::T
end

Float64 <: Real # true
Point{Float64} <: Point{Real} # false
```

- `<:` is the sub-type operator
- type hierarchy and parametric types allow to write generic code

```julia
struct Complex{T<:Real} <: Number
  re::T
  im::T
end
```

- conversion and promotion rules help to write generic code

```julia
promote_rule(::Float64, ::Int64) = Float64
+(x::Number, y::Number) = +(promote(x,y)...)
```

Types in Julia

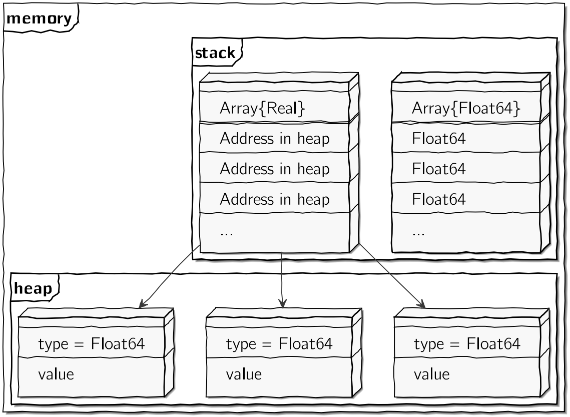

Figure 2: Memory allocation for arrays

Types in Julia

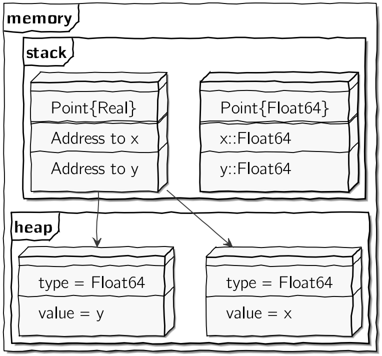

Figure 3: Memory allocation for custom types

Multiple Dispatch

Basic Principle

on a **function** call → look for types of **all** passed arguments and decide which most specific **method** to use

- types can be hidden but are always present

```
julia>  f(x, y) = 1
        f(x::Int, y) = 2
        f(x::String, y::String) = 3
f (generic function with 3 methods)
julia> f(1, 2) # calls f(x::Int64, y::Int64)
2
julia> f(1.0, 2) # calls f(x::Float64, y::Int64)
1
julia> f("a", 2) # calls f(x::String, y::Int64)
1
julia> f("a", "b") # calls f(x::String, y::String)
3
```

Main Differences to Python

Spoilsport

indexing starts with 1

- blocks closed by `end` keyword
- slice indexing includes the last element
- negative array indexes not allowed
- arrays are column major
- `"Strings"` and `'c'` characters
- more syntax[8] and functional[9] differences

[9] *MATLAB–Python–Julia cheatsheet — Cheatsheets by QuantEcon documentation*, https://cheatsheets.quantecon.org/

[9] *Noteworthy Differences from other Languages — Julia Language X.Y.Z-unknown documentation*, https://juliakorea.github.io/latest/manual/noteworthy-differences.html

Useful Packages

Instaling and Using a Pacakge

```julia
Pkg.add("NameOfPacakge")
using NameOfPacakge
```

`DataFrames`	dealing with data structures
`CSV`	reading and writing to (csv-like) files
`Plots`	API for different plotting backends like matplotlib, plotly, gr, ...
`PyPlot`	API for matplotlib, really powerful
`JuMP`	powerful API for a lot of linear and non-linear optimization tools
`ScikitLearn`	Wrapper for the Python package but parts also in Julia (and will be more in the future)
`PyCall`	call Python directly from Julia (often not needed as wrapper packages already exist)

Why Dots?

: – Symbols and Expressions[10]

express Julia language in Julia itself

```
julia> foo = "bar"
"bar"
julia> eval(:foo)
"bar"
julia> ex = :(1 + 2)
:(1 + 2)
julia> eval(ex)
3
```

[10] *Metaprogramming · The Julia Language*, https://docs.julialang.org/en/stable/manual/metaprogramming/

Why Dots?

:: – Type Annotation

explicitly tell the type

```julia
f(x::Int64) = x + 1
```

... – Slurp and Split

use variable number of arguments

```julia
f(args...) = sum(args)
f([1, 2, 3]...)
```

Why Dots?

f.() and .op – Vectorization[11]

use variable number of arguments

```
x = [1, 2, 3, 4]
x .+ 2 .* x .+ sqrt.(x)
```

- use dots for **every** operation in the expression
- allows for loop fusion into one single loop
- any function can be used without any overhead

[11] *More Dots: Syntactic Loop Fusion in Julia*, https://julialang.org/blog/2017/01/moredots

Why Dots?

Lets Try it Out!

Stanislav Kazmin[1]

[1]Max Planck Institute for Mathematics in the Sciences

May 25, 2018