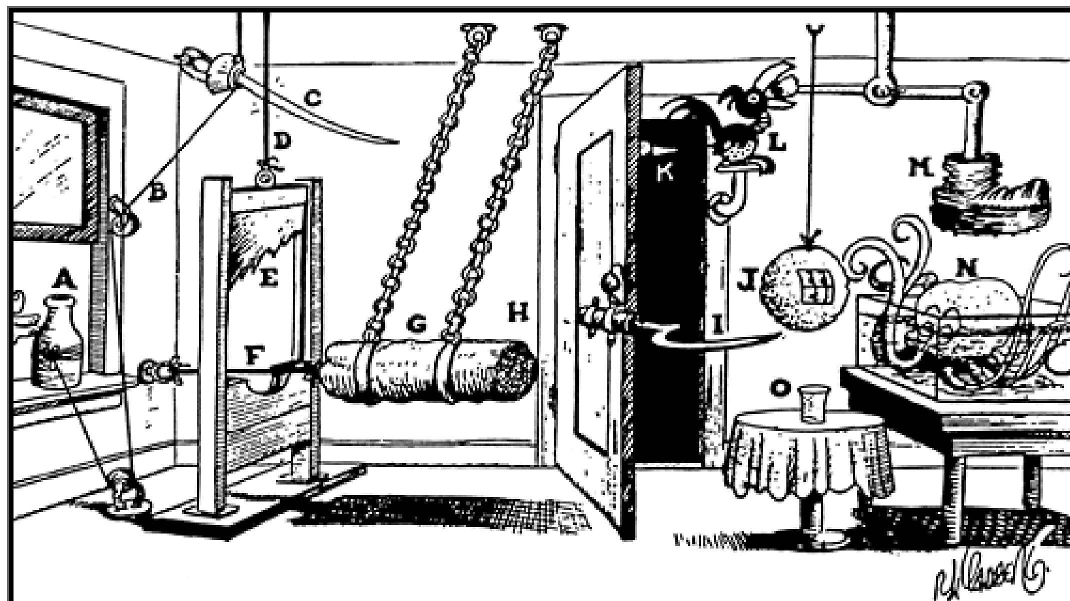
The Julia Language

Mike Nolta

Why Julia?

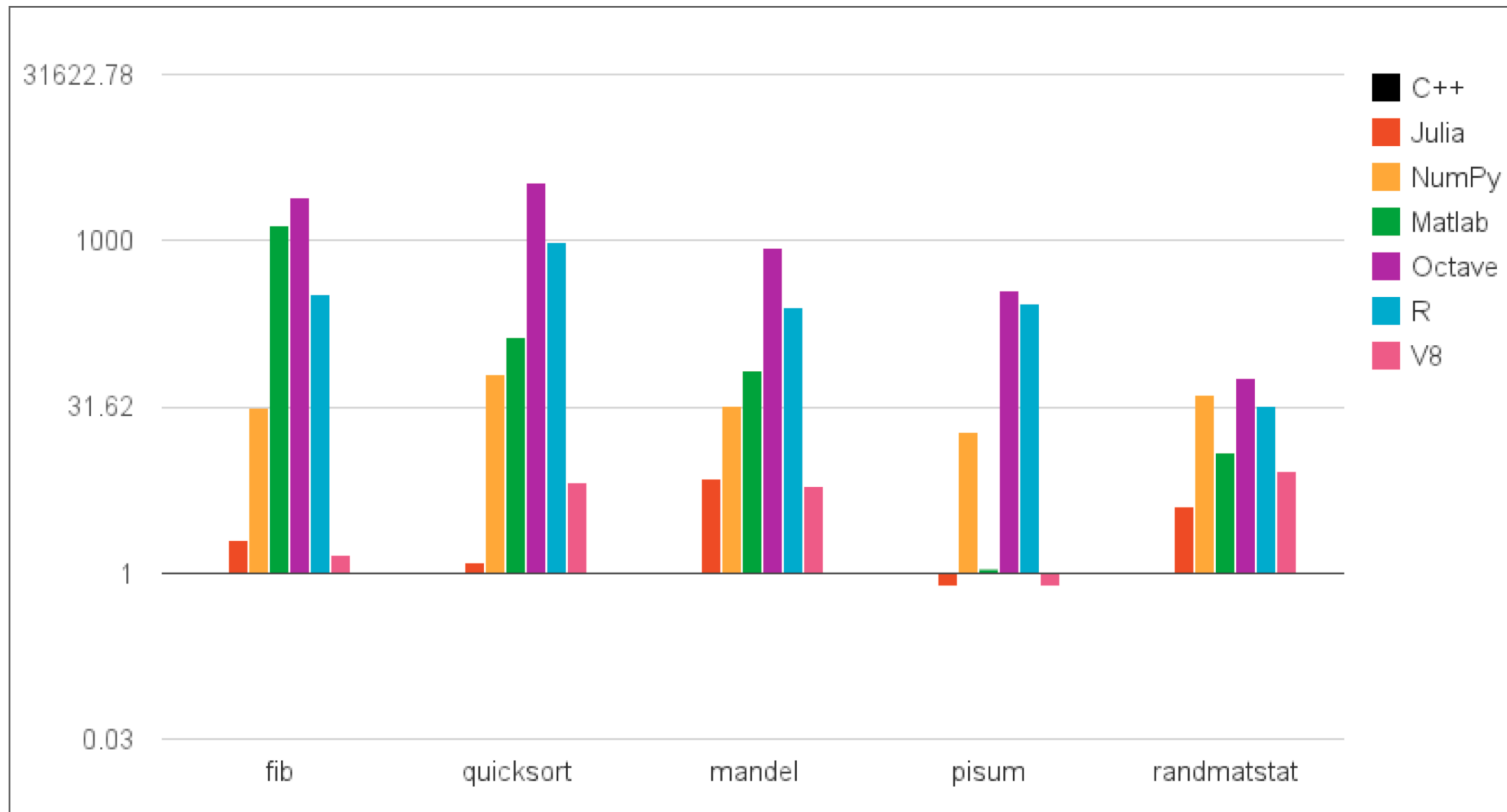


Why Julia?

- **Fast:** rich type system combined with just-in-time compiler.
- **Modern:** high-level, clean, dynamic, fast prototyping.
- **Numerical:** scientists and engineers are the target audience.
- **Free:** open source, MIT license.

Want all the best features of C/C++/Fortran, Python/Ruby/Perl, and Matlab/R/IDL combined into a single language.

Lies, damn lies, & benchmarks



Try it on the GPC

```
$ /scratch/r/rbond/nolta/julia/julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation:http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "help()" to list help topics
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.0.0+99497534.r6577
 _/ |\__'_|_|_|\__'_|  |  Commit 6577761355 (2012-10-17 13:51:20)
|__/                   |

julia> 2 + 3
5
```

ctrl-d to quit.

Numbers

```
553                 # signed integer
1_000_000           # signed integer (== 1000000)
0.                  # float
3.14156             # float
2e10                # float
0xdeadbeef          # unsigned hexadecimal
0o755               # unsigned octal
0b1011001           # unsigned binary
1//3                # rational
3.7 + 1.3im         # complex
```

Basic operations:

```
1 + 2               # 3
1 - 2               # -1
3 * 4               # 12
4 / 3               # 1.333333 (not integer division)
2 ^ 3               # 8
4 % 3               # 1
```

Unary operators

"im" is just a predefined constant, not special syntax:

```
julia> im
0 + 1im

julia> im = 4
4

julia> 2im
8
```

```
2x                    # 2*x
2.5x                  # 2.5*x
2x^3                  # 2(x^3)
2x^3y                 # 2(x^(3y))
```

Strings

```
julia> s = "a julia string"
"a julia string"

julia> name = "怒塔邁"
"怒塔邁"

julia> "my name is $name"
"my name is 怒塔邁"
```

Dictionaries

```
julia> d = { "key" => 5 }
{"key"=>5}

julia> typeof(d)
Dict{ASCIIString,Int64}

julia> d["key"]
5

julia> d["missing"]
key not found: missing
 in ref at dict.jl:395

julia> d["missing"] = 2
2

julia> d
{"missing"=>2,"key"=>5}
```

Arrays

```
julia> x = [1,2,3]
3-element Int64 Array:
 1
 2
 3

julia> x = [1 2 3]
1x3 Int64 Array:
 1  2  3

julia> x = [1. 2; 3 4]
2x2 Float64 Array:
 1.0  2.0
 3.0  4.0

julia> y = Array(Float32, 2, 3)
2x3 Float32 Array:
 7.00649e-44  1.26041e-37  4.06377e-44
 0.0          1.4013e-45   0.0
```

Arrays

```
julia> x = [1 2; 3 4]
2x2 Int64 Array:
 1  2
 3  4

julia> x[1,2]
2

julia> x[:,2]
2-element Int64 Array:
 2
 4

julia> x .< 3
2x2 Bool Array:
  true   true
 false  false
```

Conditionals

```
if a < b && (c != 5 || d >= b)
    ...
elseif some_other_condition
    ...
else
    ...
end

# ternary
x = y == 0 ? f(y) : g(y)
```

Only accepts booleans, i.e., "if p" fails if p is not a boolean.

Loops

```
for i = 1:10
    ...
end

for d = 1:0.1:10
    ...
end

for s in ["a","b","c"]
    ...
end

while x >= 0.5
    ...
end
```

Function definitions

```
function f(a, b, c)
    return g(a,b) + h(c)
end

# short form
f(a, b, c) = g(a,b) + h(c)
```

Similar to Matlab

```
 x = [1,2,3]; y = [1,2,3];

julia> [ones(size(x)) x] \ y
2-element Float64 Array:
 1.28198e-16
 1.0

julia> whos()
Base                          Module
Core                          Module
Main                          Module
x                             3-element Int64 Array
y                             3-element Int64 Array
```

Julia != Matlab

- Arrays are indexed with square brackets (e.g, A[i,j]).
- Values are passed by reference. If the callee modifies an array, the changes will be visible to the caller.
- Boolean operators (e.g., ==) always return either true or false, even when the arguments are arrays. For arrays, use the dotted operators (e.g., .==, not ==).

```
julia> a[a .< 0.5]  # instead of a(a < 0.5)
```

- Semicolons are unnecessary.

Scoping Rules

```
if some_condition
    a = 5
end
println(a)  # fails
```

```
local a
if some_condition
    a = 5
end
println(a)  # ok
```

Array Comprehensions

Instead of:

```
a = Array(Float64, 10)
for i in 1:10
    a[i] = f(i)
end
```

you can write:

```
[ f(i) for i in 1:10 ]
```

Anonymous functions

```
julia> anonfunc = (a,b) -> a + b
#<function>

julia> anonfunc(6,2)
8

julia> anonfunc = () -> 7
#<function>

julia> anonfunc()
7
```

Type system

- Dynamic: values have types, not names.
- Single inheritance (A <: B).
- Kinds:
  - Abstract types
  - Bits types
  - Composite types
  - Tuples
  - Union types
- Concrete types are final.

Abstract types

```
abstract Number
abstract Real     <: Number
abstract Integer  <: Real
abstract Signed   <: Integer
abstract Unsigned <: Integer
```

Basic types

```
Int8
Int16
Int32
Int64
Int                # Native machine size (usually Int64)
Uint8
Uint16
Uint32
Uint64
Float32
Float64
Complex64
Complex128
```

These are examples of bitstypes, defined in julia itself:

```
bitstype 64 Int64 <: Signed
```

Composite Types

```
abstract AbstractExampleType

type ExampleType <: AbstractExampleType
    a
    b::Integer
    c
end
```

```
julia> x = ExampleType("a", 1, 3.)
ExampleType("a",1,3.0)

julia> x.c
3.0

julia> ExampleType("a", 1., 3.)
no method ExampleType(ASCIIString,Float64,Float64)
 in method_missing at base.jl:70
```

Union Types

```
julia> IntOrString = Union(Int,String)
Union(String,Int64)

julia> 4::IntOrString
4

julia> "abc"::IntOrString
"abc"

julia> 3.14::IntOrString
type error: typeassert: expected Union(String,Int64), got Float64

julia> Union(Int,Integer)
Integer
```

Tuple types

```
julia> typeof((Int,Float64))
(BitsKind,BitsKind)

julia> (Int,Float64) <: (Real,Real)
true

julia> (Int,Float64) <: (Real,Integer)
false
```

Parameterized types

```
type Rational{T<:Integer} <: Real
    num::T
    den::T

    function Rational(num::T, den::T)
        if num == 0 && den == 0
            error("invalid rational: 0//0")
        end
        g = gcd(den, num)
        new(div(num, g), div(den, g))
    end
end
```

Parameterized methods

Type parameters are invariant:

```
julia> Array{Int,1} <: Array
true

julia> Array{Int,1} <: Array{Integer,1}
false
```

So to write a function which only takes integer arrays:

```
f(a::Array{Integer,1}) = ...        # wrong

f{T<:Integer}(a::Array{T,1}) = ...  # right
```

Parameterized methods

```
julia> f(a::Integer, b::Integer) = a + b

julia> f(int16(-1), uint64(1))
0x0000000000000000
```

```
julia> f{T <: Integer}(a::T, b::T) = a + b

julia> f(int16(-1), uint64(1))
no method f(Int16,Uint64)
 in method_missing at base.jl:70
```

Type aliases

```
typealias StridedMatrix{T,A<:Array}
            Union(Matrix{T},SubArray{T,2,A})
```

Multiple dispatch

These are two different functions:

```
julia> f(a::Integer, b::FloatingPoint) = "right";

julia> f(a::FloatingPoint, b::Integer) = error("wrong");

julia> f(1,1.)
"right"

julia> f(1.,1)
wrong
 in f at none:1

julia> f(1,1)
no method f(Int64,Int64)
 in method_missing at base.jl:70
```

Single dispatch

For example, the python call:

```
obj.func(arg1, ..., argN)
```

only depends on the class obj. Rewriting as

```
func(obj, arg1, ..., argN)
```

python's classes are equivalent to only dispatching on he type of the first argument.
Object-oriented code is just a special case of multiple dispatch.

"Class methods"

The equivalent of the python code:

```
class X(object):
    @classmethod
    def f(cls):
        ...
```

in julia is:

```
f(::Type{X}) = ...

# example
julia> sizeof(::Type{Int32}) = 4;

julia> sizeof(Int32)
4
```

Lowered Form

The parser converts all special syntax to functional form:

```
3 + 4       # converted to '+(3, 4)'
a[i,j]      # converted to 'ref(a, i, j)'
a[i] = x    # converted to 'assign(a, x, i)'
[a; b]      # converted to 'vcat(a, b)'
[a, b]      # converted to 'vcat(a, b)'
[a b]       # converted to 'hcat(a, b)'
```

Operators

```
//(n::Integer, d::Integer) = Rational(n,d)
//(x::Rational, y::Integer) = x.num // (x.den*y)
//(x::Integer, y::Rational) = (x*y.den) // y.num
//(x::Complex, y::Real) = complex(real(x)//y, imag(x)//y)
//(x::Real, y::Complex) = x*y'//real(y*y')
```

Iterators

The loop:

```
for i in X
    ...
end
```

gets converted to:

```
iter = start(X)
while !done(X,iter)
    i, iter = next(X,iter)
    ...
end
```

so just define the start,done,next functions to iterate over your type.

Why is julia fast?

Julia is aggressive about specializing functions based on their input types.

```
julia> f(a,b) = a + b

julia> f(1,2)
3
```

The call 'f(1,2)' causes julia to compile 'f(::Int,::Int)'.

Calling C/Fortran

You can call out to C/Fortran directly from Julia, without writing wrappers:

```
julia> ccall(:sin, Float64, (Float64,), 4.5)
-0.977530117665097

julia> sin(4.5)
-0.977530117665097
```

Genericaly:

```
lib = dlopen("libexample")
sym = dlsym(lib, :func_name)
ccall(sym, ReturnType, (ArgType1,...,ArgTypeN), Arg1, ..., ArgN)
```

Strings have type Ptr{Uint8}, pointers are Ptr{Void}.

Expressions

```
julia> ast = :(3 + 4)
:( +(3, 4) )

julia> typeof(ast)
Expr

julia> eval(ast)
7

julia> quote
       a = 3 + 4
       b = a/2
       end
quote   #  line 2:
    a = +(3, 4) #  line 3:
    b = /(a, 2)
end
```

Homoiconicity

```
julia> ast = :(3+4)
:( +(3, 4) )

julia> ast.head
call

julia> ast.args
3-element Any Array:
  +
 3
 4

julia> ast.args[1] = :*
*

julia> ast
:( *(3, 4) )

julia> eval(ast)
12
```

Expression interpolation

```
julia> a = 4
4

julia> expr(:call, {:*,:a,a})
:( *(a, 4) )

julia> :(a * $a)
:( *(a, 4) )
```

Macros

Macros are functions evaluated at compile time, whose arguments and return value are expressions.

```
julia> macro timeit(ex)
           quote
               local t0 = time()
               local val = $ex
               local t1 = time()
               println("elapsed time: ", t1-t0, " seconds")
               val
           end
       end

julia> @timeit factorial(10)
elapsed time: 0.002504110336303711 seconds
3628800

julia> @timeit factorial(10)
elapsed time: 3.0994415283203125e-6 seconds
3628800
```

Hygiene

```
julia> t0 = 6
6

julia> @timeit t0+3
elapsed time: 0.010051965713500977 seconds
1.350613586308453e9

julia> macro timeit(ex)
           quote
               local t0 = time()
               local val = $(esc(ex)) # <-------
               local t1 = time()
               println("elapsed time: ", t1-t0, " seconds")
               val
           end
       end

julia> @timeit t0+3
elapsed time: 2.1457672119140625e-6 seconds
9
```

Shelling Out

Use backticks to construct shell commands:

```
julia> run(`echo hello`)
hello

julia> readall(stdout(`echo hello`))
"hello\n"

julia> run(`echo hello` > "filename.txt")

julia> run(`cat filename.txt`)
hello
```

Shelling Out: substitutions

```
julia> s = ["a","b","c d"]
3-element ASCIIString Array:
 "a"
 "b"
 "c d"

julia> `echo $s`
`echo a b 'c d'`

julia> a = "weird'dir"
"weird'dir"

julia> `mkdir $a`
`mkdir "weird'dir"`
```

These commands are not run in a subshell, but by julia itself.

Parallel processing

Start several independent julia session on the same box:

```
$ julia -p 4
```

Run tasks in parallel:

```
julia> p = @spawn sum(randn(1000000)) # julia chooses proc
RemoteRef(1,1,1)

julia> q = @spawn sum(randn(1000000))
RemoteRef(2,1,2)

julia> fetch(p) + fetch(q)
795.6850522197668

julia> r = @spawnat 2 sum(randn(1000000)) # user chooses proc
RemoteRef(2,1,3)
```

Distributed Arrays

```
# create a distributed array w/ total size (100,200,400),
# distributed along the last dimension
A = dzeros(Float64, (100,200,400), 3)

function compute_something(A::DArray)
    B = darray(eltype(A), size(A), 3)
    for i = 1:size(A,3)
        @spawnat owner(B,i) B[:,:,i] = f(A[:,:,i])
    end
    B
end
```

Batteries included

- Linear algebra (blas/lapack)
- FFTs (fftw)
- Random numbers (mersenne twister)
- Sparse matrices (arpack, suitesparse)
- Linear programming (glpk)
- Special functions (bessel[ijkhy], airy, eta, zeta, ...)
- 2d plotting
- Regular expressions

Stdlib example: probability distributions

```
julia> load("distributions.jl")

julia> d = Distributions.Chisq(100)
Chisq(100.0)

julia> Distributions.ccdf(d, 130)
0.02351239780980862
```

Available distributions: Bernoulli, Beta, Binomial, Categorical, Cauchy, Chisq, Dirichlet, Exponential, FDist, Gamma, Geometric, HyperGeometric, Logistic, logNormal, Multinomial, NegativeBinomial, NoncentralBeta, NoncentralChisq, NoncentralF, NoncentralT, Normal, Poisson, TDist, Uniform, Weibull

Planned features

- Static compilation
- Immutable types: allow types to be passed to C as structs
- Callbacks: passing julia functions to C
- Package system (like ruby gems or cpan)

FIN

http://julialang.org