Introduction to Julia for R Users

Hans Werner Borchers
Duale Hochschule Mannheim

Köln R User Group Meetup
December 2014

New programming languages since 2002

2002	Io	Smalltalk, LISP; prototype-based
2003	Nemerle	CLI; C#-like, LISP
2003	Scala	JVM; Java, Smalltalk; stat.-typed
2004	Groovy	JVM; Java, Python, Ruby, Smalltalk
2004	Nimrod	Python, Pascal; statically-typed
2005	F# (Microsoft)	CLI; C#-like, OCaml, Haskell
2007	Clojure	JVM, CLR; LISP, Haskell, Erlang
2009	Go (Google)	C, Oberon; statically typed
2010	Rust (Mozilla)	C++-like, Erlang, LISP; LLVM
2012	Julia	MATLAB (, R); mult.dispatch; LLVM
2014	Swift (Apple)	Objective-C; LLVM

See also: Python+numba, LuaJIT, Rubinius, RLLVM, Haskell, Matlab(?), ...

The LLVM compiler infrastructure project

"The LLVM project provides libraries for a modern, industrial strength optimizer, along with code generation support [and integrated linker] for many CPUs.The libraries are build around a well specified code representation, called LLVM Intermediate Representation (IR)."

2012 ACM Software System Award

What is Julia?

“Julia is a high-level, high-performance dynamic **programming language for technical computing**, with a syntax that is familiar to users of other technical [scientific] computing environments.

“Julia’s LLVM-based just-in-time (JIT) compiler combined with the language’s design allow it to approach and often match the performance of C.

“The core of the Julia implementation is licensed under the MIT license. Various libraries used by the Julia environment include their own licenses such as the GPL, LGPL, and BSD.”

40+ scientific computing environments

APL Axiom Ch Colt[Java] Euler FreeMat GAUSS
GDL/PV-WAVE Genius gretl IDL Igor_Pro jLab
LabView Magma Maple Mathcad Mathematica MATLAB
Maxima MuPad O-Matrix Octave OriginLab Ox
PARI/GP PDL[Perl] R RLaBplus ROOT S-PLUS
SAGE SAS SCaViS SciLab SciPy[Python] SciRuby
Speakeasy Stata SciLua[LuaJIT] Yorick

REPL: "Hello, world." examples

```
» h = "Hello"; w = "world"
» println("$h, $w.")
Hello, world.

» v = [1, 2]; A = [1 2; 3 4];
» w = A * v;
» A \ w
2-element Array{Float64,1}:
 1.0
 2.0

» f(x) = x * exp(x);
» map(f, [0:0.1:1])
11-element Array{Float64,1}:
...
```

Niceties of Julia Syntax

- `a = [1.0, 2, 3]; b = a; b[1] = 0; a # 0,2,3`
- γ `= 0.57721_56649_01532_86`
- `f(x,y,z) = 2x + 3y + 4z`
- `r = 1//3 + 1//6 + 1//12 + 1//15 # 13//20`
- `factorial(big(100))`
- `H = [1/(i+j-1) for i=1:8, j=1:8]`
- `22 < pi^e < e^pi < 24 # true`
- `println("The result of pi*e is $(pi*e).")`
- `function f(x...) for a in x println(a) end end`
- **@**`time q,err = quadgk(sin, 0, pi)`
- `[1:5] |> x->x.^2 |> sum |> inv`
- `s = :bfgs # symbol`

Some differences to R

- Julia uses = for variable assignment.
- Vectors and matrices defined through brackets `[, ]`;
  matrix multiplication: `*`, operations: `.* ./ .+`;
  elementwise comparisons: `.== .<= .<` etc.
- No parentheses required in `if, for, while` constructs.
- Use `true` instead of `TRUE`; 0 or 1 are not booleans.
- Julia distinguishes scalars, vectors, matrices, or arrays by type;
  utilize type declarations for error handling.
- Function arguments are provided *by reference*, not *by value*.
- Consequence: Functions can mutate their arguments.
- Multiple return values through tuples; no lists or named vectors.
- Statistics functionality is provided in packages, not in Julia base.

Trapezoidal rule — vectorized

```
function trapz1(x, y)
  local n = length(x)
  if length(y) != n
    error("Vectors must be of same length")
  end
  sum((x[2:end]-x[1:end-1]).*(y[2:end]+y[1:end-1]))/2
end
```

```
» x = linspace(0, pi, 100); y = sin(x);
» println(trapz1(x, y)); gc()
1.9998321638939924
» @time [trapz1(x, y) for i in 1:1000];
elapsed time: 0.020384185 seconds (6921872 bytes allocated)
```

Trapezoidal rule — non-vectorized

```
function trapz2(x, y)
  local n = length(x)
  if length(y) != n
    error("Vectors 'x', 'y' must be of same length")
  end
  r = 0
  if n == 1 return r; end
  for i in 2:n
    r += (x[i] - x[i-1]) * (y[i] + y[i-1])
  end
  r / 2
end
```

```
» @time [trapz2(x, y) for i in 1:1000];
elapsed time: 0.009617445 seconds (3215904 bytes allocated)
```

Trapezoidal rule — type-stable

```
function trapz3(x, y)
  local n = length(x)
  if length(y) != n
    error("Vectors 'x', 'y' must be of same length")
  end
  r = 0.0
  if n == 1 return r; end
  for i in 2:n
    r += (x[i] - x[i-1]) * (y[i] + y[i-1])
  end
  r / 2
end
```

```
@time [trapz3(x, y) for i in 1:1000];
elapsed time: 0.001451867 seconds (47904 bytes allocated)
```

Trapezoidal rule — w/o bounds checking

```
function trapz{T<:Number}(x::ArrayT,1, y::ArrayT,1)
  local n = length(x)
  if length(y) != n
    error("Vectors 'x', 'y' must be of same length")
  end
  r = zero(T)
  if n == 1 return r end
  for i in 2:n
    @inbounds r += (x[i] - x[i-1]) * (y[i] + y[i-1])
  end
  r / 2
end
```

```
» @time [trapz(x, y) for i in 1:1000];
elapsed time: 0.000730233 seconds (47904 bytes allocated)
```

Trapezoidal rule — comparisons

Results and comparison with R and Python

```
          Timings      Result                        μs/loop
trapz1  0.020384185  1.9998321638939924              20.4
trapz2  0.009617445  1.9998321638939929               9.6
trapz3  0.001451867  1.9998321638939929               1.5
trapz   0.000740450  1.9998321638939929               0.75

R:        unvect.  285    μs,  vect.   19 μs
 comp:              78    μs,          15 μs  (= Renjin?)
 Rcpp:    unvect.    3.5  μs                  (inline)
Python:   unvect.  119    μs,  vect.   39 μs
 numba:   unvect.    0.72 μs,  vect.   54 μs

MATLAB:   unvect.   12    μs,  vect.   35 μs
Octave:   unvect. 2000    μs,  vect.  200 μs
```

Performance tips

- Avoid global variables (or make them `const`).
- For best performance, use **non-vectorized code**; devectorize array assignments, write explicite loops, etc.
- Break functions into multiple definitions, based on types.
- **Type stability**: Avoid changing the type of a variable.
- Access arrays in memory order, i.e., along columns.
- Avoid arrays with abstract type parameters: `Vector{Real}`
- Pay attention to **memory allocations** (see *macro* `@time`):
  - preallocate larger data structures (arrays);
  - avoid the need to copy data structures.
- Apply performance *annotations* if appropriate (e.g., `@inbounds`)

Julia's numerical types

```
Number
    Real
        FloatingPoint
            BigFloat
            Float64 Float32 Float16
        Integer
            BigInt
            Signed
                Int128  Int64 [=Int=] Int32  Int16  Int8
            Unsigned
                Uint128 Uint64 Uint32 Uint16 Uint8
            Bool
            Char
        Rational
    Complex
```

Operator overloading

```
» methods(+)  # 146 methods for generic function +
 +(x::Bool) at bool.jl:34
 +(x::Bool,y::Bool) at bool.jl:37
 +(y::FloatingPoint,x::Bool) at bool.jl:47
 ...
»  +(s, t) = s * t                  # would be wrong
» ++(s, t) = s * t                  # is not possible
» ⊕(s, t)  = s * t                  # is not advisable

»  +(s::String, t::String) = s * t
»  "123" + "..." + "xyz"            #=> "123...xyz"
»  +("123", "...", "xyz")
»  +(["123", "...", "xyz"]...)
```

User-defined (or: composite) types

```
immutable GaussInt <: Number  # or: type GaussInt
    a::Int
    b::Int
    # GaussInt(n::Int, m::Int) = new(n, m)
end
GaussInt(1, 1)  #=> GaussInt(1,1)

import Base.show, Base.norm, Base.isprime
show(io::IO, x::GaussInt) = show(io, complex(x.a, x.b))
GaussInt(1, 1)  #=> 1 + 1im

*(x::GaussInt, y::GaussInt) =
        GaussInt(x.a*y.a - x.b*y.b, x.a*y.b + x.b*y.a);
norm(x::GaussInt) = x.a^2 + x.b^2;
isprime(x::GaussInt) = isprime(norm(x));  # wrong!
```

Optimization packages in Julia

- **Optim** – BFGS, CG, simulated annealing
- **GLPK, Cbc, Clp** – mixed-integer linear programming
- **CPLEX, Gurobi, Mosek** – interfacing commercial systems
- **Ipopt** – interface to the IPOPT nonlinear solver (COIN-OR)
- **NLopt** – interface to the NLopt nonlinear optimization library
- **ECOS, Convex** – (disciplined) convex programming solvers
- **JuMP, MathProgBase** – optimization modeling languages
- **BlackBoxOptim, JuliaCMAES** – global optimization
- **LsqFit, MinFinder** – least-squares, all minima

JuMP – Julia for Mathematical Programming

- Domain-specific modeling language for mathematical programming (i.e., optimization)
- Syntax mimics natural mathematical expressions
- Problem classes: LP, MILP, SOCP, nonlinear programming
- Generic, solver-independent user interface
- Supported solvers:
  Cbc, Clp, CPLEX, ECOS, GLPK, Gurobi, Ipopt, MOSEK, NLopt
- Speed: Problem creation faster than commercial modeling tools (AMPL, GAMS, etc.)

Modeling example: Knapsack problem

```
» p = [92, 57, 49, 68, 60, 43, 67, 84, 87, 72];
» w = [23, 31, 29, 44, 53, 38, 63, 85, 89, 82];
» cap = 165; nitems = 10;

» using JuMP, Cbc
» m = Model( solver=CbcSolver() )
» @defVar( m, x[1:nitems], Bin )
» @setObjective( m, Max, sum{p[i]*x[i], i=1:nitems})
» @addConstraint(m, sum{w[i]*x[i], i=1:nitems} <= cap)

» status = solve(m)
» getObjectiveValue(m)                      # 165
» idx = [getValue(x[i]) for i in 1:nitems]
[1,1,1,0,0,0,0,0,0,1]                           # 1,2,3,10
```

Automatic Differentiation (AD)

Automatic differentiation "is a set of techniques to numerically evaluate the derivative of a function specified by a computer program."
Example: **lambertW** is an iteratively defined function computing the *Lambert W* special function, the reverse of $x \to x \cdot e^x$.

```
» lambertW(1.0)   # 0.5671432904097838 Omega const.

» # numerical derivative at 1.0
» using DualNumbers
» lambertW(dual(1.0, 1.0))
0.5671432904097838 + 0.3618962566348892du

» # exact derivative
» 1.0 / (1 + lambertW(1.0) / exp(lambertW(1.0))
0.3618962566348892
```

Statistics packages in Julia

- StatsBase, Distributions
- Distances, Clustering
- HypothesesTests, KernelDensity
- DimensionalityReduction
- DataArrays, DataFrames
- GLM (*Doug Bates*)
- MCMC

- MLBase
- NMF, RegERMs
- SVM, NeuralNets

DataArrays and DataFrames

```
» using RDatasets      # 700+ R data sets
» planets = dataset("HSAUR", "planets")
» planets[:Mass]

» using DataArrays     # NA support
» using DataFrames
» describe(planets)    # summary
```

The **DataFrames** package supports functionality like the following :

- join, split-apply-combine
- sorting, reshaping
- factors, model frames (formulae)

Calling C and Fortran

Shared library `specfun.so` has been generated with the R command “`R CMD SHLIB specfun.f`”

```
» x = 0.5
» y = ccall(
            (:gamma_,"./specfun"), # (function, library)
            Float64,               # type of return value
            (Ptr{Float64}, ),      # input types as tuple
            &x );                  # input(s)
» y
1.772453850905516                  # sqrt(pi)
```

But: The Julia Core team intends to make possible the compilation of Julia functions and packages into shared libraries!
BUT: ...

R calling Julia?

"Julia has a nice and simple C interface. So that gets us something like .C(). But as recently discussed on r-devel, you really do not want .C(), in most cases you rather want .Call() in order to pass actual SEXP variables representing real R objects. So right now I see little scope for Julia from R because of this limitation.
Maybe an indirect interface using tcp/ip to Rserve could be a first start before Julia matures a little and we get a proper C++ interface. [...]
And the end of the day, some patience may be needed. I started to look at R around 1996 or 1997 when Fritz Leisch made the first announcements on the comp.os.linux.announce newsgroup. And R had rather limited facilities then (**but the full promise of the S language, of course, so we knew we had a winner**). [...]
Julia may well get there. But for now I suspect many of us will get work done in R, and have just a few curious glimpses at Julia."
Dirk Eddelbüttel, Stackoverflow, April 1, 2012

Parallelization

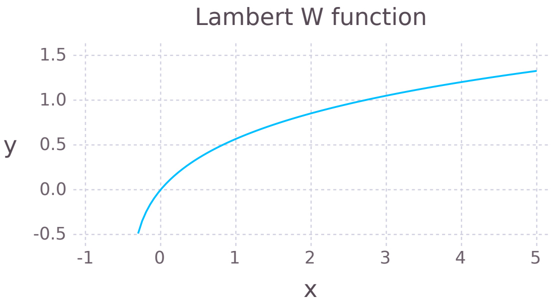
"Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple [processors] in separate memory domains at once."

```
$ julia -p 2
...
» r = remotecall(1, rand, 2, 2)
» fetch(r)
» @spawn rand(2, 2)
» s = @spawn rand(2, 2)

» @everywhere f(x) = x * exp(x)
» r1 = remotecall_fetch(1, f, 1)
» r2 = remotecall_fetch(2, f, 2)
```

Grammar of Graphics in Julia

```
» using Gadfly
» xs = linspace(-0.3, 5.0); ys = map(lambertW, xs);
» fig = plot(x=xs, y=ys, Geom.line,
                Guide.title="Lambert W function")
» draw(PDF("gadfly.pdf", 4inch, 2inch), fig)
```



Calling Python

Example: Function interpolation, symbolic integration

```
» using PyCall
» xs = [1.0:10]; ys = sqrt(xs);
» @pyimport scipy.interpolate as spi
» fpy = spi.interp1d(xs, ys, kind="cubic")
» pycall(fpy, Float64, pi)  # 1.7723495528382518

» using SymPy
» x,y,z = Sym("x y z")
» limit(sin(x)/x, x, 1)     # 1
» z = integrate(sin(x)/x, x, 1, Inf)
-Si(1) + 1.5707963267949
» float(z)
0.6247132564277136
```

Web Resources

- **Julia home page**: julialang.org
- **Source Code**: github.com/JuliaLang/julia
  Personal Package Archives: /juliareleases [0.3], /julianightlies [0.4]
- **Available packages**: http://iainnz.github.io/packages.julialang.org/
- **Julia Manual**: http://docs.julialang.org/en/release-0.3/manual/
- **Mailing List**:
  https://groups.google.com/forum/?fromgroups=#!forum/julia-users
- **Julia Blogroll**: http://www.juliabloggers.com/

The Julia Manual is a quite reasonable introduction to the Julia language.

David Sanders: Julia tutorial, SciPy 2014
Steven Johnson: Keynote talk, EuroSciPy 2014

Editors for Julia development

- **Julia Studio** [outdated, comm.?]
- **Light Table** (w/ Jewel/Juno plugin)
- **IPython notebook** (w/ IJulia)
  see the *Jupyter* project
- Editors with syntax highlighting (and auto-completion):
  Sublime Text 3 (w/ Sublime-Julia) [Linux]
  TextMate [Mac], gedit or Kate [Linux]
  Notepad++ [Windows]
- Eclipse (w/ LiClipse)
  Emacs (w/ julia-mode.el)
  vim (w/ julia-vim)

Julia Special Interest Groups

Special Interest Groups (SIGs) in Julia have a function similar to the 'Task Views' in R, but they also kind of *organize* the task area.

- **JuliaOpt** – mathematical optimization
- **JuliaStat** – statistics and machine learning
- **JuliaQuant** – quantitative finance
- **JuliaDiff** – differentiation tools
- **JuliaDB** – database integration

- BioJulia, JuliaAstro, JuliaQuantum
- JuliaSparse, JuliaGPU, JuliaWeb

Conclusions / Questions

1. Will Julia survive and become a mayor player?
2. . . .
3.
4.
5.
6.
7.
8.
9.
10.