Julia

A Fast Dynamic Language for Technical Computing

Created by: Jeff Bezanson, Stefan Karpinski, Viral B. Shah & Alan Edelman
A Fractured Community

Technical work gets done in many different languages

- C, C++, R, Matlab, Python, Java, Perl, Fortran, ...

Different optimal choices for different tasks

- statistics ➞ R
- linear algebra ➞ Matlab
- string processing ➞ Perl
- general programming ➞ Python, Java
- performance, control ➞ C, C++, Fortran

Larger projects commonly use a mixture of 2, 3, 4, ...
One Language

We are **not** trying to replace any of these

- C, C++, R, Matlab, Python, Java, Perl, Fortran, ...

What we are trying to do:

- **allow developing complete technical projects in a single language**
  without sacrificing productivity or performance

This does *not* mean not using components in other languages!

- Julia uses C, C++ and Fortran libraries extensively
“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.”
Collapsing Dichotomies

Many of these are just a matter of design and focus

‣ stats vs. linear algebra vs. strings vs. glue vs. metaprogramming

The hardest dichotomy to bridge:

‣ high-level, dynamism, productivity
‣ low-level, efficiency, performance

High-level languages traditionally use a split model

‣ R/Python/Matlab for high-level coding
‣ C/C++/Fortran for low-level coding
Leverage and Control

Fortunately, it’s not the 1990’s anymore

‣ LLVM provides an incredible just-in-time compilation infrastructure

Julia uses LLVM and aggressive JIT to bridge high/low schism

‣ requires deep reconsideration of language design to take advantage

Gives unprecedented control and leverage with ease-of-use

‣ do low-level tricks previously only possible in C or assembly
‣ call C/Fortran libraries trivially and efficiently
**Julia in a Nutshell**

Dynamically typed

- with performance like static languages

Sophisticated parametric type system

- but you never have to use it (no performance penalty)

Matlab-like syntax (simplified), easy to learn and use

- but homoiconic like Lisp, with real macros, metaprogramming, etc.

Broad-spectrum, highly polymorphic

- “a+b” can do a single machine instruction or start up a cluster
function qsort!(a,lo,hi)
    i, j = lo, hi
    while i < hi
        pivot = a[(lo+hi)>>>1]
        while i <= j
            while a[i] < pivot; i = i+1; end
            while a[j] > pivot; j = j-1; end
            if i <= j
                a[i], a[j] = a[j], a[i]
                i, j = i+1, j-1
            end
        end
    end
    if lo < j; qsort!(a,lo,j); end
    lo, j = i, hi
end
return a
end
function randmatstat(t,n)
    v = zeros(t)
    w = zeros(t)
    for i = 1:t
        a = randn(n,n)
        b = randn(n,n)
        c = randn(n,n)
        d = randn(n,n)
        P = [a b c d]
        Q = [a b; c d]
        v[i] = trace((P'*P)^4)
        w[i] = trace((Q'*Q)^4)
    end
    std(v)/mean(v), std(w)/mean(w)
end
function copy_to(dst::DArray, src::DArray)
    @sync begin
        for p in dst.pmap
            @spawnat p copy_to(localize(dst), localize(src, dst))
        end
    end
    return dst
end

function copy_to(dest::AbstractArray, src)
    i = 1
    for x in src
        dest[i] = x
        i += 1
    end
    return dest
end
Multiple Dispatch

Some basic rules for addition of “primitives”

\[ + (x::\text{Int64}, y::\text{Int64}) = \text{boxsi64}(\text{add}\_\text{int}(x,y)) \]
\[ + (x::\text{Float64}, y::\text{Float64}) = \text{boxf64}(\text{add}\_\text{float}(x,y)) \]

The `promote` function (defined in Julia) converts to common type

\[ \text{promote}(1,1.5) \Rightarrow (1.0,1.5) \]

With a few generic rules like this, numeric promotion Just Works™

\[ + (x::\text{Number}, y::\text{Number}) = + (\text{promote}(x,y)...) \]
Multiple Dispatch

```plaintext
function +{S,T}(A::Array{S}, B::Array{T})
    P = promote_type(S,T)
    S = promote_shape(size(A),size(B))
    F = Array(P,S)
    for i = 1:numel(A)
    end
    return F
end
```
for f in (:+, :-, :.*, :div, :mod, :&, :|, :$)
    @eval begin
        function ($f){S,T}(A::Array{S}, B::Array{T})
            P = promote_type(S,T)
            S = promote_shape(size(A),size(B))
            F = Array(P,S)
            for i = 1:numel(A)
                F[i] = ($f)(A[i], B[i])
            end
            return F
        end
    end
end
Calling C/Fortran Libraries

Load the library and use “ccall” with the function signature:

```julia
getpid() = ccall(:getpid, Uint32, ())

system(cmd) = ccall(:system, Int32, (Ptr{Uint8},), cmd)

libfdm = dlopen("libfdm")
besselj0(x) =
   ccall(dlsym(libfdm,:j0), Float64, (Float64,), x)

function fill!(a::Array{Uint8}, x::Integer)
   ccall(:memset, Void, (Ptr{Uint8},Int32,Int),
         a, x, length(a))
   return a
end
```
Calling LibRmath

libRmath = dlopen("libRmath")

dt(x, p1, give_log) =
    ccall(dlsym(libRmath,:dt),
          Float64, (Float64,Float64,Int32),
          x, p1, give_log)

pt(x, p1, give_log) =
    ccall(dlsym(libRmath,:pt),
          Float64, (Float64,Float64,Int32),
          x, p1, give_log)

dt(x, p1) = dt(x, p1, false)
pt(x, p1) = pt(x, p1, false)
libpython = dlopen("libpython")

call(dlsym(libpython,:Py_Initialize), Void, ())

call(dlsym(libpython,:PyRun_SimpleString),
    Int32, (Ptr{Uint8},),
    "print 'Hello from Python.'")

# later...
ccall(dlsym(libpython,:Py_Finalize), Void, ())
Some Low-Level Hackery

Find the first float after a given value that “misbehaves”

```plaintext
function find_x_times_inv_x_neq_1(x)
    while x*(1/x) == 1
        x = nextfloat(x)
    end
    return x
end
```

The “nextfloat” function is defined as

```plaintext
nextfloat(x::Float64) = boxf64(add_int(x,1))
```
### Performance

<table>
<thead>
<tr>
<th>Function</th>
<th>Julia 3f670da0</th>
<th>Python 2.7.1</th>
<th>Matlab R2011a</th>
<th>Octave 3.4</th>
<th>R 2.14.2</th>
<th>JavaScript V8 3.6.6.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>1.97</td>
<td>31.47</td>
<td>1336.37</td>
<td>2383.80</td>
<td>225.23</td>
<td>1.55</td>
</tr>
<tr>
<td>parse_int</td>
<td>1.44</td>
<td>16.50</td>
<td>815.19</td>
<td>6454.50</td>
<td>337.52</td>
<td>2.17</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.49</td>
<td>55.84</td>
<td>132.71</td>
<td>3127.50</td>
<td>713.77</td>
<td>4.11</td>
</tr>
<tr>
<td>mandel</td>
<td>5.55</td>
<td>31.15</td>
<td>65.44</td>
<td>824.68</td>
<td>156.68</td>
<td>5.67</td>
</tr>
<tr>
<td>pi_sum</td>
<td>0.74</td>
<td>18.03</td>
<td>1.08</td>
<td>328.33</td>
<td>164.69</td>
<td>0.75</td>
</tr>
<tr>
<td>rand_mat_stat</td>
<td>3.37</td>
<td>39.34</td>
<td>11.64</td>
<td>54.54</td>
<td>22.07</td>
<td>8.12</td>
</tr>
<tr>
<td>rand_mat_mul</td>
<td>1.00</td>
<td>1.18</td>
<td>0.70</td>
<td>1.65</td>
<td>8.64</td>
<td>41.79</td>
</tr>
</tbody>
</table>

**Figure:** benchmark times relative to C++ (smaller is better).
Project Statistics

Hundreds of popular numerical functions

Getting traction as an open-source project:

- 510,000+ page views
- 125,000+ visitors
- 6,000+ downloads
- 1,300+ GitHub followers
- 50+ contributors
- 4+ Stefans

http://julialang.org/