A Fast Dynamic Language for Technical Computing

Stefan Karpinski, Jeff Bezanson, Viral B. Shah & Alan Edelman

Two Language Problem

People love dynamic environments

- for data analysis and exploration
- but dynamism and performance are at odds

A standard compromise:

- high-level logic in convenient, dynamic language (Matlab, Python, R)
- performance-critical code in static, low-level language (C, C++, Fortran)

Creates a huge barrier to development

- continually breaking the abstraction barrier indicates a poor abstraction

Core Design

- dynamically typed

  but with highly descriptive type system

- generic functions

  multiple dispatch is the central paradigm

- aggressively specialized on run-time types

  (technically an implementation detail, but an important one)

  “Everything is a template”

Some Features

- Lisp-style macros
- cooperative threading (coroutines)
- async I/O + distributed parallel computing primitives
- easy C/Fortran calling
  - no wrapper code, no compilation
  - callbacks from C to Julia code
- great LAPACK and FFTW interfaces
  - it's not just LU any more
- native layout structs and struct arrays
- portable – runs on 64/32 OS X, Linux & Windows

Matlab-like

```
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
```

Low-Level

```
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
        if lo < j; qsort!(a,lo,j); end
        lo, j = i, hi
    end
    return a
end
```

High-Level

```
function copy_to(dst::DArray, src::DArray)
    @sync begin
        for p in dst.pmap
            @spawnat p copy_to(localize(dst),
                               localize(src,dst))
        end
        return dst
end
```

Modular Integers

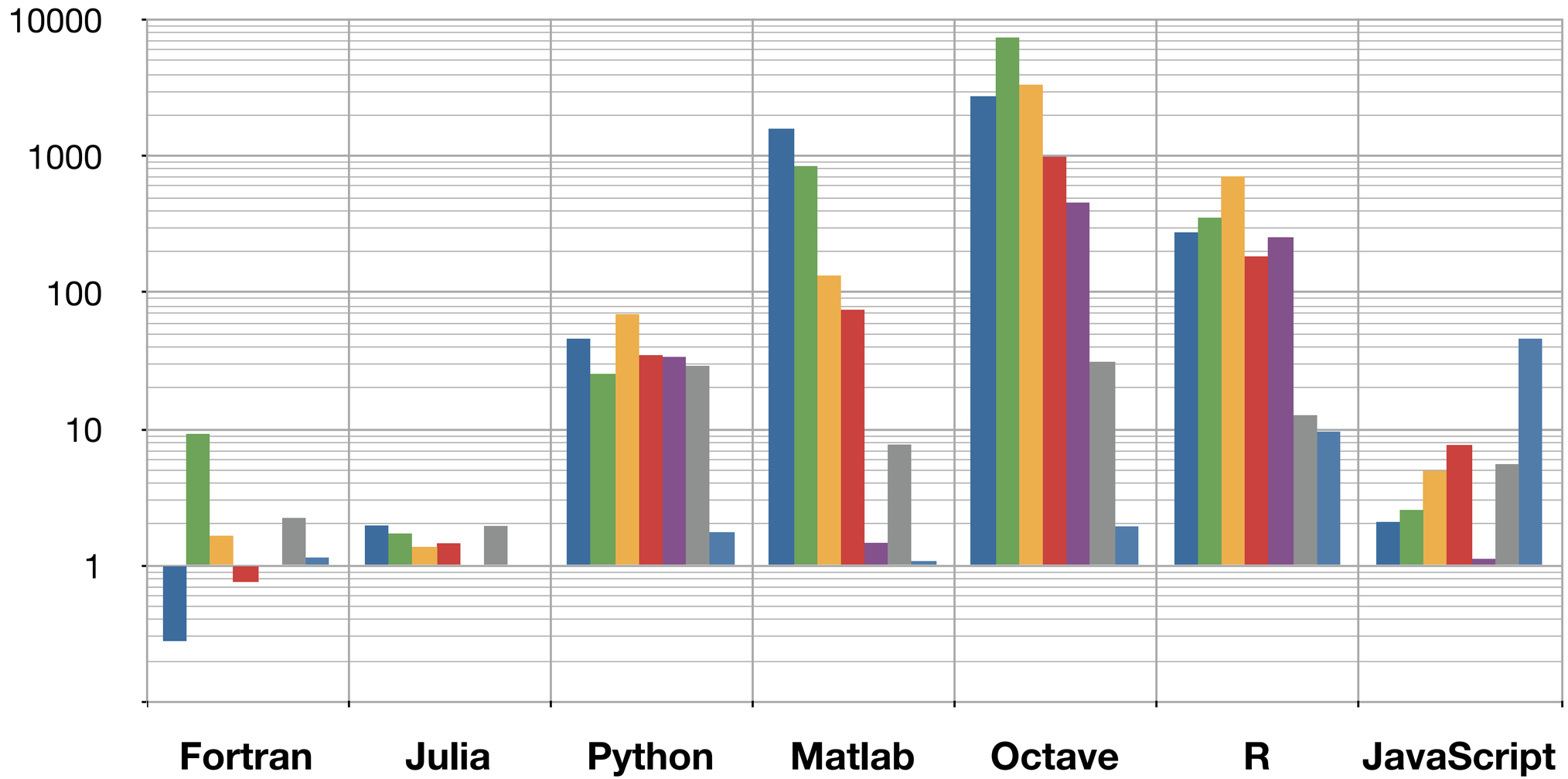
```
immutable ModInt{n} <: Integer
    k::Int
    ModInt(k) = new(mod(k,n))
end

-{n}(a::ModInt{n}) = ModInt{n}(-a.k)
+{n}(a::ModInt{n}, b::ModInt{n}) = ModInt{n}(a.k+b.k)
-{n}(a::ModInt{n}, b::ModInt{n}) = ModInt{n}(a.k-b.k)
*{n}(a::ModInt{n}, b::ModInt{n}) = ModInt{n}(a.k*b.k)

convert{n}(::Type{ModInt{n}}, i::Int) = ModInt{n}(i)
promote_rule{n}(::Type{ModInt{n}}, ::Type{Int}) = ModInt{n}

show{n}(io::IO, k::ModInt{n}) = print(io, "$(k.k) mod $n")
showcompact(io::IO, k::ModInt) = print(io, k.k)
```

Obligatory Performance Slide



execution time relative to C++ (lower is better)

Reports from the Real World

"[R]eports … indicate that Julia gives rather significant boosts over Matlab/R, sometimes by even more than the benchmarks might suggest. That was surprising to me, since I expected the gap to be largest for benchmarks.

…

[O]ne common factor was fairly sizable (but not ridiculous) memory requirements; perhaps Julia's ability to manage memory in a more fine-grained fashion pays major dividends for such problems."

– Tim Holy, WUSTL
http://goo.gl/r6qwz

Simplex Benchmarks

source: Miles Lubin & Iain Dunning

`https://github.com/mlubin/SimplexBenchmarks`

- Benchmark of some important operations:

	Julia	C++	C++bnd	Matlab	PyPy	Python
Sp. mat-sp. vec	1.29	0.90	1.00	5.79	19.20	417.16
Sp. vector scan	1.59	0.96	1.00	13.98	13.81	48.39
Sp. `axpy`	1.85	0.70	1.00	19.12	9.21	78.65

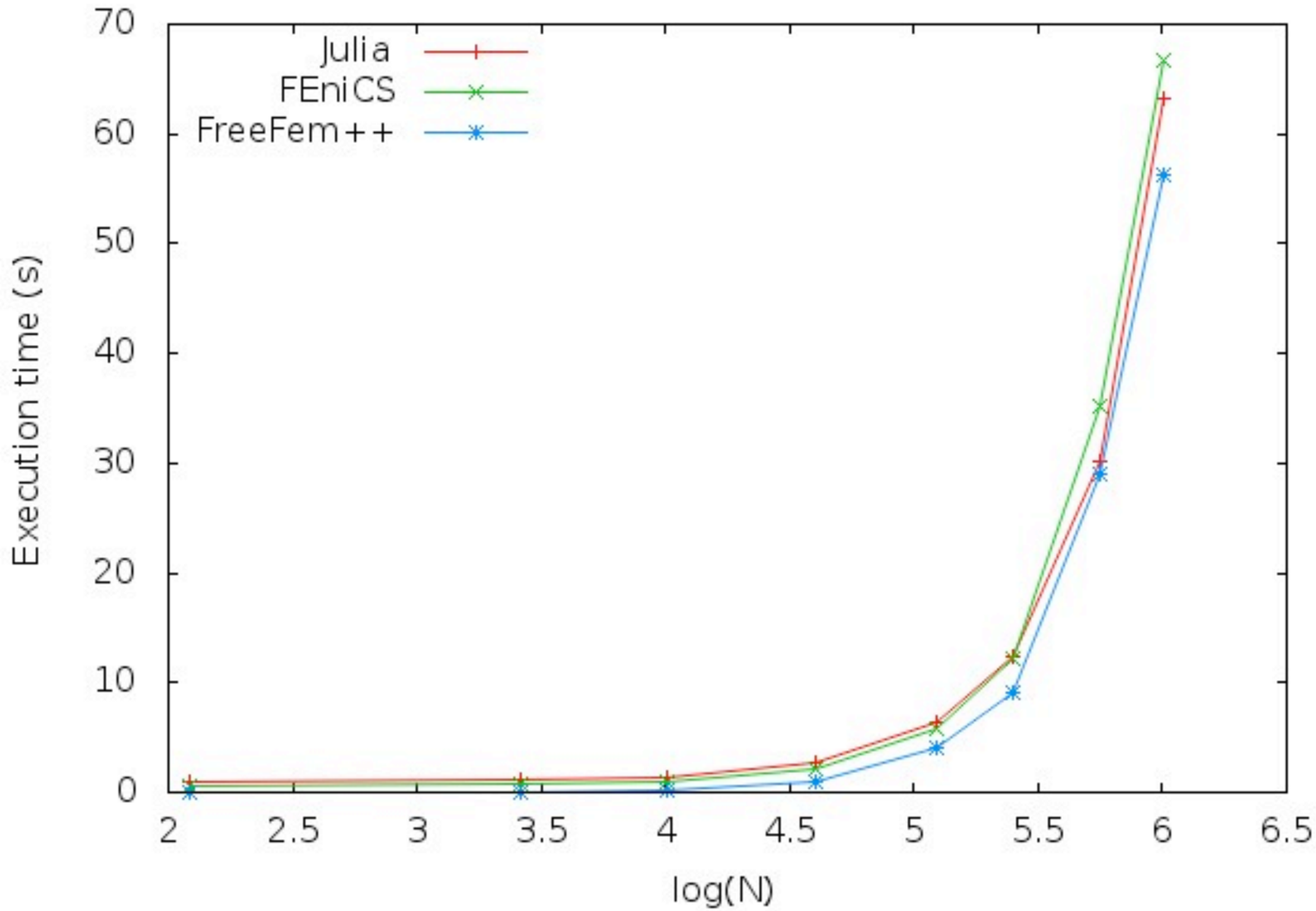
- C++bnd = C++ with bounds checking
- Execution times relative to C++bnd

Finite Element Programming

Comparison by Amuthan Ramabathiran [http://goo.gl/SRciE]:

- FEniCS: “collection of software for high level finite element code development written in Python and C++”
- FreeFem++: “partial differential equations solver written in C++ with its own DSL (Domain Specific Language) with a C++ like syntax.”
- a simple Julia FEM solver: “Thanks to Julia’s elegant syntax the code is largely self-explanatory.”

Finite Element Programming



Finite Element Programming

"[W]hat is really interesting about Julia is the relative ease with which various strategies can be implemented and tested without leading to code swell, while at the same time resulting in high performance code.

...

Julia appears to be a very good choice for developing research oriented finite element software that is both fast and easy to develop."

– Amuthan Ramabathiran
http://goo.gl/SRciE

Memory Control

Three key features:

- C-like, C-compatible “struct” types (and immutable versions too)
- efficiently laid out typed Arrays
- in-place APIs for operating on data

  sort! filter! lufact! cholfact!

Community

130+ contributors to base Julia

1000+ mailing list subscribers

140+ packages, including:

BloomFilters Cairo Calculus Calendar Catalan Clp Clustering CoinMP ContinuedFractions Cubature DICOM DataFrames DataStructures DecisionTree Devectorize DimensionalityReduction Distance Distributions DualNumbers Elliptic FITSIO FastaRead GLFW GLM GLPK GLUT GSL GeometricMCMC GoogleCharts Graphs Grid Gtk Gurobi HDF5 HDFS HTTP Hadamard HypothesisTests ICU Images ImmutableArrays Ito JSON JudyDicts Jyacas KLDivergence LIBSVM Languages LazySequences LinProgGLPK Loss MAT MATLAB MCMC MDCT MLBase MNIST MarketTechnicals MathProg MathProgBase Meshes MixedModels Mongo Mongrel2 NHST NLopt ODBC ODE OpenGL OpenSSL Optim PLX Polynomial Profile PyCall Quandl RDatasets RNGTest RandomMatrices Resampling Rif Rmath Roots SDE SDL SVM SemidefiniteProgramming SimJulia Sims Stats Sundials SymbolicLP TOML TextAnalysis TextWrap TimeModels TimeSeries Tk TopicModels TradingInstrument Trie UTF16 Units WAV ZMQ kNN