Julia

A Fast Dynamic Language for Technical Computing

Jeff Bezanson & Stefan Karpinski

Why Julia?

Dynamic languages are extremely popular for numerical work:

- Matlab, R, NumPy/SciPy, Mathematica, etc.
- very simple to learn and easy to do research in

However, all have a “split language” approach:

- high-level dynamic language for scripting low-level operations
- C/C++/Fortran for implementing fast low-level operations

Libraries in C — no productivity boost for library writers

Forces vectorization — sometimes a scalar loop is just better

Three Key Features

Sophisticated *dynamic* type system exposed in the language

- dependent parametric types; abstract types, a.k.a. traits
- polymorphism of all kinds — ad hoc, parametric & duck

Multiple dispatch as the core unifying paradigm

- when well-implemented, fast & ubiquitous, it is *qualitatively* different
- many features can be seen as special cases of multiple dispatch

One language for a broad spectrum of programming levels

- `a*b` can compile down to a single machine instruction
- `a*b` can start a computation on a cluster of 1000s of machines

Low-Level Code

```
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
```

Medium-Level Code

```
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
```

High-Level Code

```
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
```

Calling C/Fortran

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

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


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


function fill!{T<:Union(Int8,Uint8)}(a::Array{T}, x::Integer)
    ccall(:memset, Void, (Ptr{T},Int32,Int), a, x, length(a))
    return a
end
```

The Numeric Promotion Dilemma

Most languages allow you to mix numeric types

- not having this gets annoying very quickly

  Ada, OCaml (?), assembly

Traditional solution is to build promotion rules into the language

- otherwise too slow
- but doesn't work for user-defined types

Ideally want something generic, extensible & fast

```
1 + 2.5      0.5 + 3im     ...
```

Promotion via Multiple Dispatch

Built-in definitions:

```
function promote{T,S}(x::T, y::S)
    P = promote_type(T,S)
    convert(P,x), convert(P,y)
end

+(x::Number, y::Number) = +(promote(x,y)...)
```

When adding a new type:

```
promote_rule(::Type{Complex128}, ::Type{Float64}) = Complex128

convert(::Type{Complex128}, x::Real) = complex128(x,0)

+(z::Complex128, w::Complex128) =
     complex128(z.re+w.re, z.im+w.im)
```

Type System — What's Normal

Nominative type hierarchy

Bits types, composite types, abstract types

Tuple types (argument lists)

Union types

Types have parameters (invariant)

```
Rational{Int32}(1,2)

Rational(1,2)
```

Type System — What's Unusual

Parametric methods and "type patterns"

```
r{T<:Integer}(x::T, y::T) = Rational{T}(x,y)
r(1,2)
```

Singleton kinds

```
sizeof(::Type{Int16}) = 2
sizeof(Int16) ⇒ 2
```

Example — matrices that can be passed to LAPACK:

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

Dynamic Type Inference

Tags, not types

Tries to “guess” the tags

Entirely run-time semantics

Can improve the algorithm without updating the spec

Dynamic Type Inference

Abstract interpretation of lowered form:

- assignments, calls, conditional branches, exception handlers

Apply type transfer functions to handle calls

Small set of primitives with simple, known t-functions

The t-function for generic functions is

$$T(f, t_{\mathrm{arg}}) = \bigsqcup_{(s,g)\in f} T(g, t_{\mathrm{arg}} \sqcap s)$$

Key Optimizations

Aggressive method specialization

Lots of inlining

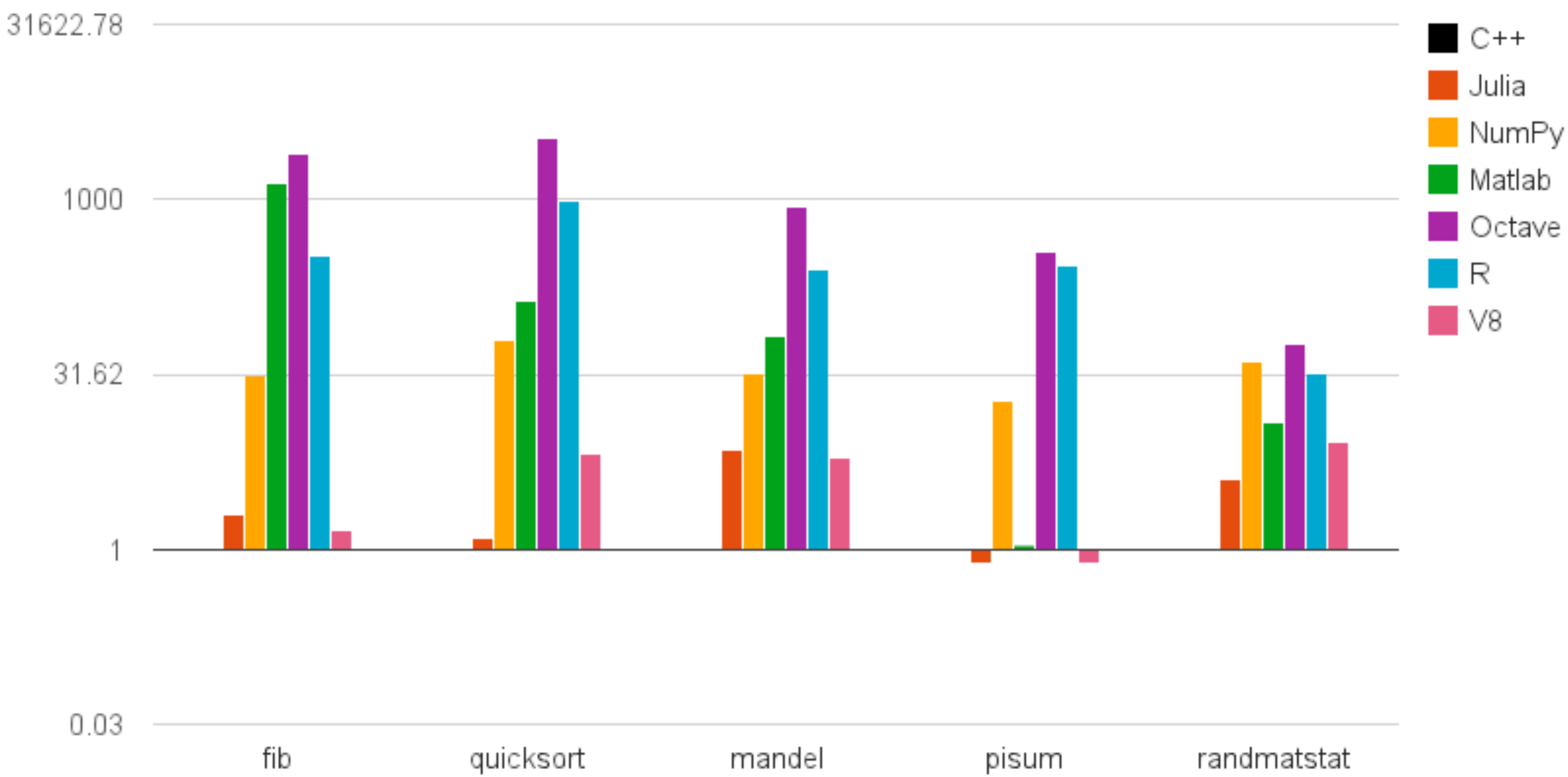
Elimination of `apply()`

```
apply(f, (a, b))    ⇒  f(a,b)
apply(f, t::(T,S))  ⇒  f(t[1], t[2])
```

Multiple value cons-elimination

```
(a, b) = (f(), g()) ⇒  t1 = f(); t2 = g()
                        a = t1;   b = t2
```

Performance



Disadvantages

Method ambiguities

- can print a very specific warning (using type intersection)

Generated code, compiler data structures and type information take up memory

- realistically, can't run Julia in < 200Mb today

About 144 bytes/LOC in the library

Building from scratch is slow

- ~15 sec system image build time to prime the cache (but done off-line)

Modularity is a bit tricky with multiple dispatch

Type info only flows "forward" — no return type overloading

People Like It!

"Frustrated matlab and R user wanting a language that doesn't sacrifice performance."

"Where has Julia been this past two years!? I had searched for it high and low, day and night, to the point of nearly driving myself insane."

"I'm having a lot of *fun* (productive fun!) using Julia and hope to be able to contribute."

"...everything I wished I'd had in MATLAB and for data analysis for years now…"

"I'm really excited that you're building a language that looks very much like what I've wanted for over ten years now."

Project Statistics

Hundreds of popular numerical functions

Getting traction as an open-source project:

- 300,000+ page views
- 100,000+ visitors
- 3,000+ downloads
- 1,000+ GitHub followers
- 40+ contributors
- 4+ Stefans