Julia

A Fast Dynamic Language for Technical Computing

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

Two-Tier Architectures

Standard compromise between convenience and performance

- high-level logic in a dynamic language
- heavy lifting in C and Fortran

Pragmatic for many applications, but has drawbacks

- prefer to write compute-intensive code at high-level too
- forces vectorization — often unnatural, lots of temporaries
- complexity — mediation between type domains, gc schemes
- significant overhead, makes whole-program optimization difficult
- social barrier — makes contributing to internals daunting

Fast & Dynamic

These days, dynamic languages can be fast

- PyPy
- LuaJIT
- JavaScript V8

What if you designed a language with this knowledge?

- take maximal advantage of fast techniques (JIT, type inference, etc.)
- provide more expressiveness at the same time (types, dispatch)

Julia's Approach

Fast, dynamic, and expressive

- generic functions — i.e. dynamic multiple dispatch
- rich type system — expressive and aids type inference
- speed allows Julia's library to be written in Julia itself

Notes:

- we use LLVM for code generation — but not a silver bullet
- type inference is *not* Hindley-Milner — data-flow based, dynamic

Why Types?

There are always types!

- values in “untyped” languages still have types
- there just isn’t a way to *talk* about types

Fast implementations have type systems

- to get good performance, you need to know about types
- why not make a good type system and expose it?

OTOH, in Julia you never *need* to mention a type

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
```

Multiple Dispatch

Some basic rules for addition of "primitives"

```
+(x::Int64, y::Int64)     = boxsi64(add_int(x,y))
+(x::Float64, y::Float64) = boxf64(add_float(x,y))
```

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

```
promote(1,1.5) => (1.0,1.5)
```

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

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

Fancy Method Signatures

```
typealias LapackElt Union(Float64,Float32,Complex128,Complex64)
typealias StridedMatrix{T,A<:Array} Union(Matrix{T},SubArray{T,2,A})

function *{T<:LapackElt}(A::StridedMatrix{T},X::StridedVector{T})

  # shenanigans to call LAPACK...

end

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

Hacking the Core is Easy

Newcomers have added **major** pieces within weeks of using Julia

- BitArrays
- SubArrays
- Distributions
- DataFrames (in progress)

What requires major surgery in many systems is easy

- changing core arithmetic behaviors — e.g. overflow, promotion
- adding new “bits types”; new string types

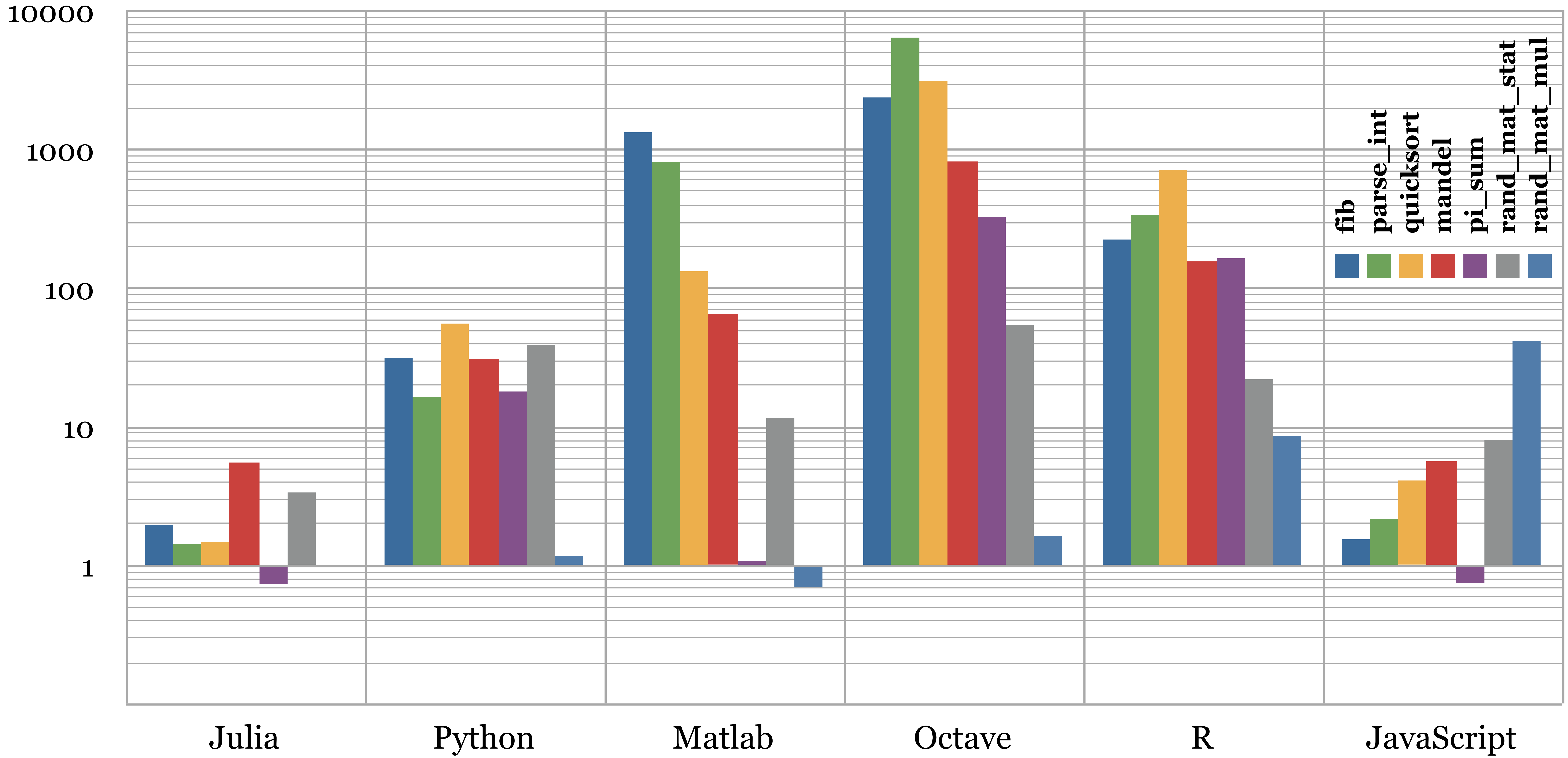
Changing Integer Promotions

```
promote_rule(::Type{Uint8} , ::Type{Int8} ) = UintInt
promote_rule(::Type{Uint8} , ::Type{Int16}) = UintInt
promote_rule(::Type{Uint8} , ::Type{Int32}) = UintInt
promote_rule(::Type{Uint8} , ::Type{Int64}) = UintInt64

promote_rule(::Type{Uint16}, ::Type{Int8} ) = UintInt
promote_rule(::Type{Uint16}, ::Type{Int16}) = UintInt
promote_rule(::Type{Uint16}, ::Type{Int32}) = UintInt
promote_rule(::Type{Uint16}, ::Type{Int64}) = UintInt64

if WORD_SIZE == 64
    promote_rule(::Type{Uint32}, ::Type{Int8} ) = Int
    promote_rule(::Type{Uint32}, ::Type{Int16}) = Int
    promote_rule(::Type{Uint32}, ::Type{Int32}) = Int
else
    promote_rule(::Type{Uint32}, ::Type{Int8} ) = Uint
    promote_rule(::Type{Uint32}, ::Type{Int16}) = Uint
    promote_rule(::Type{Uint32}, ::Type{Int32}) = Uint
end
promote_rule(::Type{Uint32}, ::Type{Int64}) = UintInt64
```

Performance



Python Interop

Still nascent, but lots of ideas

- call statically-compiled Julia code via C ABI
- share on-disk data formats
- call libpython from Julia:

```
libpython = dlopen("libpython")
ccall(dlsym(libpython,:Py_Initialize), Void, ())
ccall(dlsym(libpython,:PyRun_SimpleString),
      Int32, (Ptr{Uint8},),
      "print 'Hello from Python.'")
```

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.”