Julia

A Fast Dynamic Language for Technical Computing

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

“Technical Computing”

Matlab Maple Mathematica SciPy SciLab IDL R Octave S-PLUS SAS J APL Maxima Mathcad Axiom Sage Lush Ch LabView O-Matrix PV-WAVE Igor Pro OriginLab FreeMat Yorick GAUSS MuPad Genius SciRuby Ox Stata JLab Magma Euler Rlab Speakeasy GDL Nickle gretl ana

Three Features

- dynamic language
- sophisticated parametric type system
- multiple dispatch

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
```

Distributed

```
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
```

Four Constraints

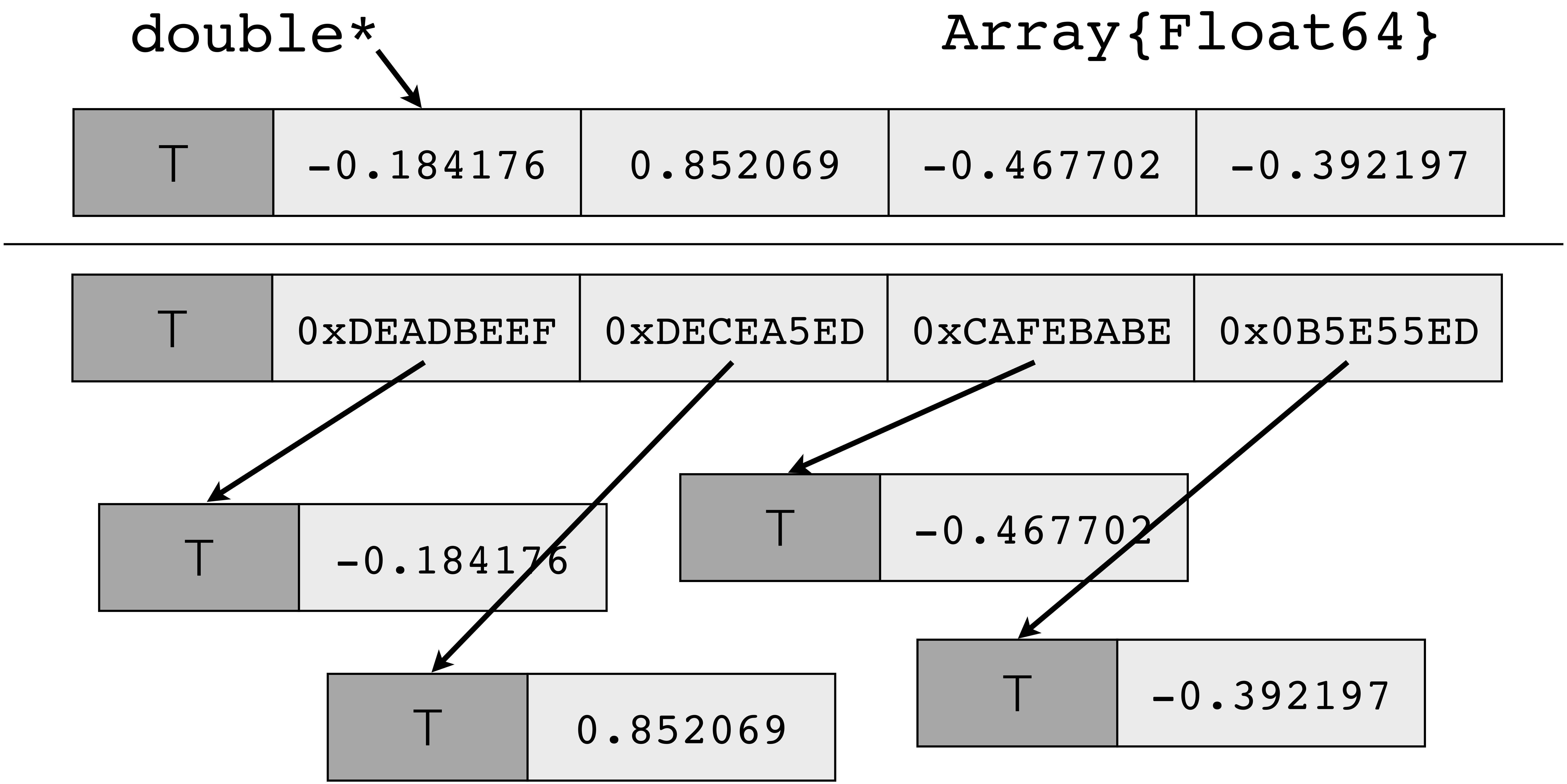
general

- interactive, productive, tangible
- unified type system

numerical

- efficient arrays of numbers
- math operators are just functions

Arrays



double*
Array{Float64}
T
-0.184176
0.852069
-0.467702
-0.392197
T
0xDEADBEEF
0xDECEA5ED
0xCAFEBABE
0x0B5E55ED
T
-0.184176
T
-0.467702
T
0.852069
T
-0.392197

Plus

C

```
MyInt add(MyInt a, MyInt b)
MyFloat add(MyInt a, MyFloat b)
MyFloat add(MyFloat a, MyInt b)
```

C++

```
MyInt operator+(MyInt a, MyInt b)
MyFloat operator+(MyInt a, MyFloat b)
MyFloat operator+(MyFloat a, MyInt b)
MyNumber operator+(MyNumber a, MyNumber b)
```

Plus

Ruby

```
class MyInt < MyNumber
  def +(b)
    ...
  end
end

MyInt + 1
1 + MyInt
```

Python

```
class MyInt(MyNumber):
  def __add__(b):
    ...
  def __radd__(b):
    ...
```

Plus

Julia

```
+(a::MyInt, b::MyInt) = ...
+(a::MyInt, b::MyFloat) = ...
+(a::MyFloat, b::MyInt) = ...

+(a::Int, b::MyInt) = ...
+(a::MyInt, b::Int) = ...
```

Promotion

Some basic rules for addition of “primitives”

```
+(x::Int64, y::Int64)     = box(Int64,add_int(x,y))
+(x::Float64, y::Float64) = box(Float64,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)...)
```

Technical Computing

Matlab Maple Mathematica SciPy SciLab IDL R Octave S-PLUS SAS J APL Maxima Mathcad Axiom Sage Lush Ch LabView O-Matrix PV-WAVE Igor Pro OriginLab FreeMat Yorick GAUSS MuPad Genius SciRuby Ox Stata JLab Magma Euler Rlab Speakeasy GDL Nickle gretl ana

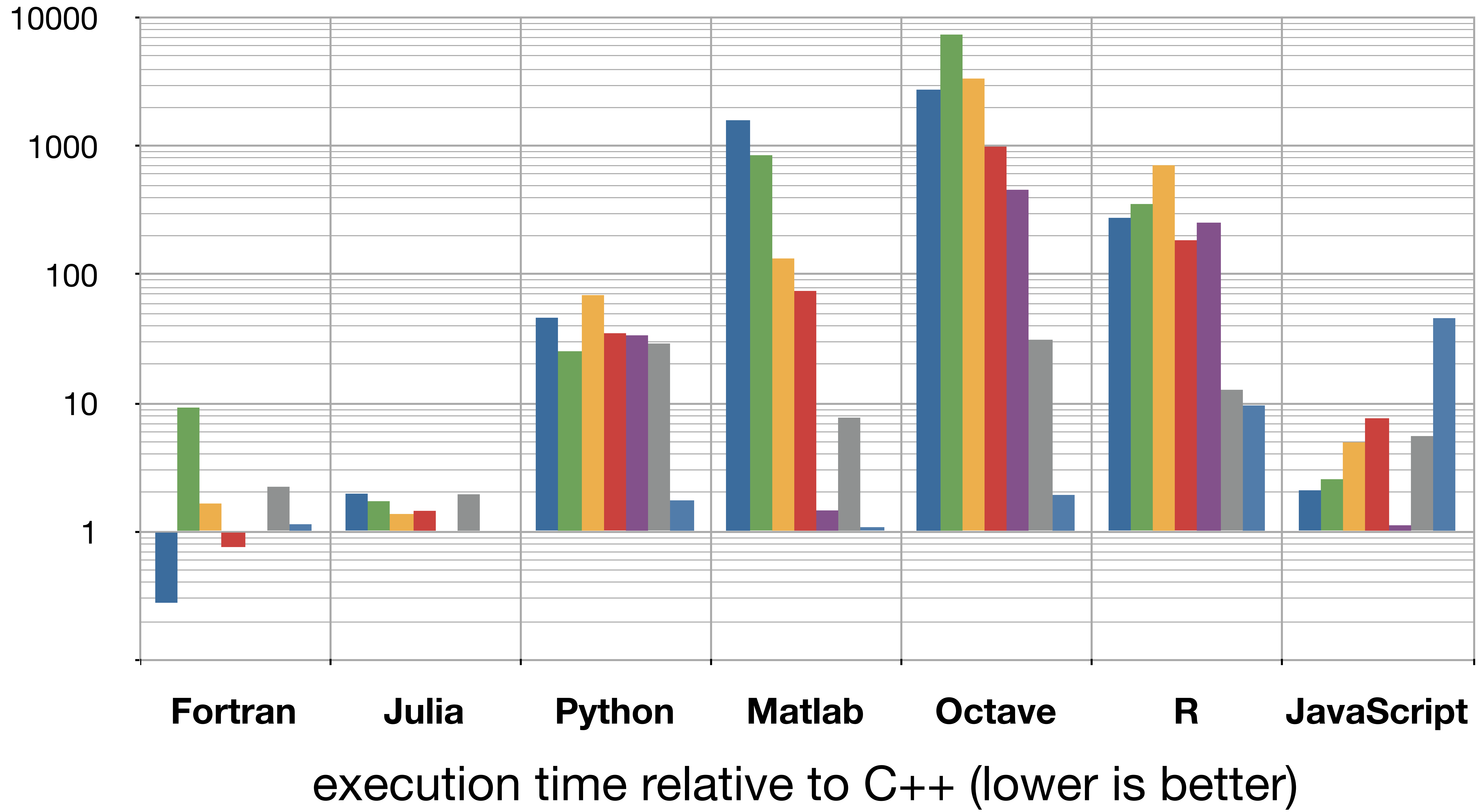
Changing Integer Promotions

```
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



ModInt

```
type ModInt{N} <: Integer
    k::Int
    ModInt(k) = new(k % N)
end

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