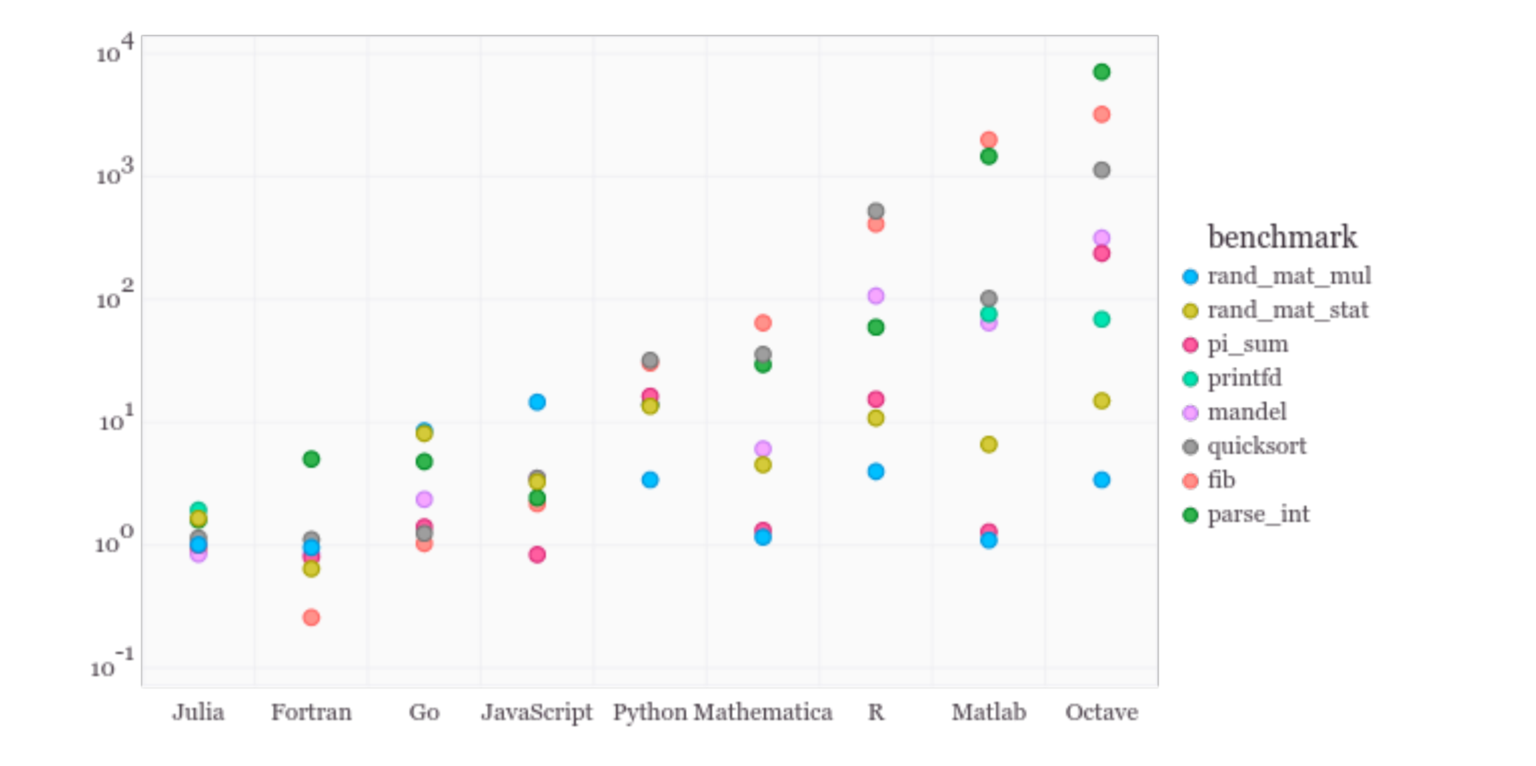
How Julia Goes Fast

Leah Hanson



10^4
10^3
10^2
10^1
10^0
10^{-1}
Julia
Fortran
Go
JavaScript
Python
Mathematica
R
Matlab
Octave
benchmark
rand_mat_mul
rand_mat_stat
pi_sum
printfd
mandel
quicksort
fib
parse_int

	Fortran gcc 4.8.2	Julia 0.3.2	Python 2.7.6	R 3.1.1	Matlab R2014a	Octave 3.8.1	Mathe-matica 10.0	JavaScript V8 3.14.5.9	Go go1.2.1	LuaJIT gsl-shell 2.3.1	Java 1.7.0_65
fib	0.70	2.39	79.95	553.57	4638.29	9764.56	163.43	3.73	2.14	2.38	0.90
parse_int	4.88	1.93	12.24	53.23	1580.52	9106.83	17.66	2.33	3.77	6.79	5.55
quicksort	1.31	1.24	33.23	255.73	54.43	1766.13	48.21	2.91	1.11	2.36	1.69
mandel	0.74	0.72	12.18	54.06	51.23	391.25	6.24	1.55	0.99	0.71	0.57
pi_sum	0.99	1.06	16.93	16.55	1.27	279.53	1.51	2.19	1.33	1.18	1.00
rand_mat_stat	1.15	2.14	19.04	16.65	10.48	35.92	6.71	3.32	8.92	4.34	4.01
rand_mat_mul	4.73	1.11	1.24	1.91	1.18	1.25	1.21	17.19	9.83	1.44	2.35

Figure: benchmark times relative to C (smaller is better, C performance = 1.0).

Main Points

1. Design choices make Julia fast.
2. Design and implementation choices work together.
3. You should try using Julia.

1. What problem is Julia solving?
2. What design choices does that lead to?
3. How does the implementation make it fast?

What problem are we solving?

Julia is for scientists.

(and also programmers)

Non-professional programmers who use programming as a tool.

What do they need in a language?

- Easy to learn, easy to use.
- Good for writing small programs and scripts.
- Fast enough for medium to large data sets.
- Fast, extensible math, especially linear algebra.
- Many libraries, including in other languages.

Easy and Fast

with lots of library support

How is Julia better than what they already use?

i.e. Numpy

The Two Language Problem

i.e. C and Python

Two Language Problem

You learn Python, and use Numpy.

Fast Numpy code is in C, so you have to learn that to contribute.

Fast Julia code is in Julia, so domain experts can write fast Julia libraries.

Julia has to be both C and Python

The Big Decisions

Static-dynamic trade-offs.

Static, compiled, fast

Dynamic, interpreted, easy

Implementation

Compiled:

- Compile-time
- Run native code
- No REPL

Interpreted:

- No compile-time
- Running parsed code
- Full REPL

Design

Static:

- Static typing
- Static dispatch

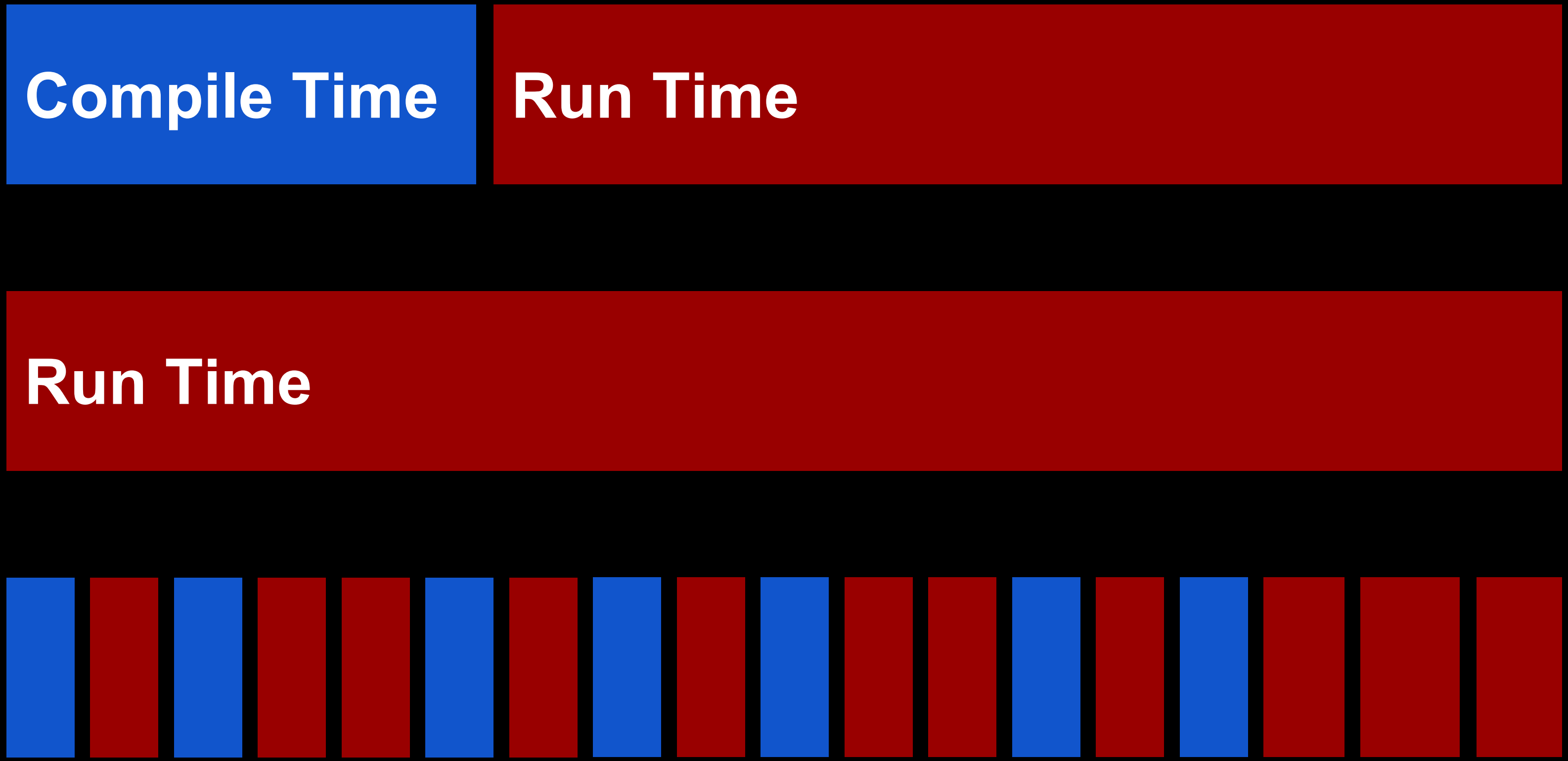
Dynamic:

- Dynamic typing
- Dynamic dispatch

Specific Julia Design Choices

- JIT Compilation (implementation)
- Sort-of Dynamic Types (language)
- Dynamic Multiple Dispatch (language)

JIT Compilation



Compile Time
Run Time
Run Time

Our compiler needs to be fast.

But it has access to run-time information.

The Type System

- **Values** have types.
- Variables are informally said to have the same type as the value they contain.

```
x = 5
x = “hello world”
```

- **Values** have types.
- Variables are informally said to have the same type as the value they contain.

```
x = 5::Int64
x = “hello world”::String
```

- **Values** have types.
- Variables are informally said to have the same type as the value they contain.

```
x = 5
x = “hello world”
```

Concrete Types

- Can be instantiated (i.e. you can make one)
- Determine layout in memory
- **Types cannot be modified after creation**
- One supertype; **no subtypes**

```
type ModInt
  k::Int64
  n::Int64
end
```

Multiple Dispatch

Multiple Dispatch

- Named functions are generic
- Each function has one or more methods
- Each method has a specific argument signature and implementation

```
x = ModInt(3,5)
x + 5
5 + x
```

```
function Base.+(m::ModInt, i::Int64)
  return m + ModInt(i, m.n)
end

function Base.+(i::Int64, m::ModInt)
  return m + i
end
```

```
class ModInt
  def +(self, i::Int64)
    self + ModInt(i, self.n)
  end
end

# monkey patch Base for Int64 + ModInt?
```

Haskell Type Classes

The Details

JIT Compilation & Multiple Dispatch

JIT-ed Multiple Dispatch

1. Intersect possible method signatures and inferred argument types
2. Generate code for that

JIT-ed Multiple Dispatch

1. Intersect possible method signatures and inferred argument types
2. Generate code for that

```
foo(5)
foo(6)
foo(7)
```

With Caching

1. Check method cache for function & inferred argument types. (If it’s there, skip to step 4.)
2. If not, intersect possible method signatures and inferred argument types.
3. Generate code for that method and the inferred argument types.
4. Run the generated code.

JIT Compilation & Types

```julia
function Base.*(n::Number, m::Number)
  if n == 0
    return 0
  elseif n == 1
    return m
  else
    return m + ((n - 1) * m)
  end
end
```

Calling The Function

```
4 * 5 # => 20
4.0 * 5.0 # => 20.0
```

Generic Functions

Aggressive Specialization

Code size vs. Speed

Dispatch is Slow

So we should avoid it!

```
function a(n)
  result1 = b(n)
  n += result1
  r2 = b(n)
  return n + r2
end
```

```
function b(n)
  return n + 2
end

function b(n::Int64)
  return n * 2
end
```

In-Lining

the copy-paste approach

Devirtualization

write down the IP to avoid DNS

Issue #265

function a ignores updates to function b

Boxed/Unboxed

Unboxed:

- Just the bits
- Compiler knows type
- Could be on stack or heap or in register

Boxed:

- type tag + bits
- Compiler needs the tag to know the type
- Stored on the heap

A Tale of Two Functions

```
function a()
  sum = 0
  for i=1:100
    sum += i/2
  end
  return sum
end
```

```
function b()
  sum = 0.0
  for i=1:100
    sum += i/2
  end
  return sum
end
```

Let's Time Them

```
julia> @time a()
elapsed time: 9.517e-6 seconds (3248
bytes allocated)
2525.0

julia> @time b()
elapsed time: 2.285e-6 seconds (64
bytes allocated)
2525.0
```

WHOA! Look at those bytes!

```
julia> @time a()
elapsed time: 9.517e-6 seconds (3248
bytes allocated)
2525.0

julia> @time b()
elapsed time: 2.285e-6 seconds (64
bytes allocated)
2525.0
```

Unstable Types and the Heap

Non-concrete types means you must allocate the boxed value on the heap.

Boxed immutable types mean you must make a new copy on the heap for each change.

This type instability leads to a lot of allocations.

julia> code_native(a,())

```
	.section	__TEXT,__text,regular,pure_instructions
Filename: none
Source line: 2
	push	RBP
	mov	RBP, RSP
	push	R15
	push	R14
	push	R13
	push	R12
	push	RBX
	sub	RSP, 56
	mov	QWORD PTR [RBP - 80], 6
Source line: 2
	movabs	RAX, 4308034112
	mov	RCX, QWORD PTR [RAX]
	mov	QWORD PTR [RBP - 72], RCX
	lea	RCX, QWORD PTR [RBP - 80]
	mov	QWORD PTR [RAX], RCX
	mov	QWORD PTR [RBP - 56], 0
	mov	QWORD PTR [RBP - 48], 0
	movabs	RAX, 4328810048
Source line: 2
	mov	QWORD PTR [RBP - 64], RAX
	mov	EBX, 1
	mov	R15D, 10000
Source line: 4
	movabs	R12, 4295395472
	movabs	R13, 4328736592
	movabs	RCX, 4416084224
	movsd	XMM0, QWORD PTR [RCX]
	movsd	QWORD PTR [RBP - 88], XMM0
	movabs	R14, 4295030048
	mov	QWORD PTR [RBP - 56], RAX
	call	R12
	mov	QWORD PTR [RAX], R13
	xorps	XMM0, XMM0
	cvtsi2sd	XMM0, RBX
	mulsd	XMM0, QWORD PTR [RBP - 88]
	movsd	QWORD PTR [RAX + 8], XMM0
	mov	QWORD PTR [RBP - 48], RAX
	movabs	RDI, 4362376736
	lea	RSI, QWORD PTR [RBP - 56]
	mov	EDX, 2
	call	R14
Source line: 3
	inc	RBX
Source line: 4
	dec	R15
	mov	QWORD PTR [RBP - 64], RAX
	jne	-70
Source line: 6
	mov	RCX, QWORD PTR [RBP - 72]
	movabs	RDX, 4308034112
	mov	QWORD PTR [RDX], RCX
	add	RSP, 56
	pop	RBX
	pop	R12
	pop	R13
	pop	R14
	pop	R15
	pop	RBP
	ret
```

julia> code_native(b,())

```
       .section       __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 4
       push   RBP
       mov    RBP, RSP
       xorps  XMM0, XMM0
       mov    EAX, 1
       mov    ECX, 100
       movabs RDX, 4416084592
       movsd  XMM1, QWORD PTR [RDX]
Source line: 4
       xorps  XMM2, XMM2
       cvtsi2sd       XMM2, RAX
       mulsd  XMM2, XMM1
       addsd  XMM0, XMM2
Source line: 3
       inc    RAX
Source line: 4
       dec    RCX
       jne    -28
Source line: 6
       pop    RBP
       ret
```

Macros for speed?

Macros

Julia has Lisp-style macros.

Macros are evaluated at compile time.

Macros should be used sparingly.

But how can they make code faster?

What is Horner's Rule?

$ax^2 + bx + c$ = a*x*x + b*x + c

Too many multiplies!

a*x*x + b*x + c = (a*x + b)*x + c

What is Horner's Rule?

$ax^3 + bx^2 + cx + d$

```
= a*x*x*x + b*x*x + c*x + d
= (a*x + b)*x*x + c*x + d
= ((a*x + b)*x + c)*x + d
= d + x*(c + x*(b + x*a))
```

Horner’s Rule as a Macro

```
# evaluate p[1] + x * (p[2] + x * (....)),
# i.e. a polynomial via Horner's rule
macro horner(x, p...)
  ex = esc(p[end])
  for i = length(p)-1:-1:1
    ex = :($(esc(p[i])) + t * $ex)
  end
  return Expr(:block, :(t = $(esc(x))), ex)
end
```

What does calling it look like?

```
@horner(t,
         0.14780_64707_15138_316110e2,
        -0.91374_16702_42603_13936e2,
         0.21015_79048_62053_17714e3,
        -0.22210_25412_18551_32366e3,
         0.10760_45391_60551_23830e3,
        -0.20601_07303_28265_443e2,
         0.1e1)
```

Is it fast?

See PR#2987, which added `@horner`

Used to implement the function `erfinv` for finding the inverse of the error function for real numbers.

4x faster than Matlab
3x faster than SciPy

which both call C/Fortran libraries

Is it plausible?

The compiled Julia methods will have inlined constants, which are very optimizable.

A reasonable way to implement it in C/Fortran would involve a (run-time) loop over the array of coefficients.

Conclusion

Main Points

1. Design choices make Julia fast.
2. Design and implementation choices work together.
3. You should try using Julia.

P.S.

Julia is a fun, general-purpose language that you should try! :)

Leah Hanson

@astrieanna

blog.LeahHanson.us

Leah.A.Hanson@gmail.com