Why

?

Motivation and comparison to other languages

Oliver Schulz
Max Planck Institute for Physics
oschulz@mpp.mpg.de

MAX-PLANCK-GESELLSCHAFT

Max-Planck-Institut für Physik
(Werner-Heisenberg-Institut)

Julia for HEP Mini-workshop, September 27th 2021

Why Julia?

Science needs code - but how to write it?

- Choice of programming language(s) matter!
- Need to balance:
    - Learning time
    - Productivity
    - Performance
- Usually involves compromises

Programming Language Options

- C++:
    - Pro: Very fast (in expert hands)
    - Pro: Really cool new concepts (even literally) in C++11/14/17/...
    - Con: Complex, takes long time to learn and much longer to master
    - Con: Straightforward tasks often result in lengthy code
    - Con: No memory management (General protection faults)
    - Con: No universal package management
    - Con: Composability isn't great
    - Con: C++ job market now focused on true experts

Programming Language Options

- Python:
    - Pro: Broad user base, popular first programming language
    - Pro: Easy to learn, good standard library
    - Con: Can't write time-critical loops in Python, workarounds like Numba/Cython have many limitations, don't compose well
    - Con: Language itself fairly primitive, not very expressive
    - Con: Duck-Typing necessitates lots of test code
    - Con: No effective multi-threading
    - Con: Composability isn't great

What else is there?

- Fortran:
    - Pro: Math can be really fast
    - Con: Old language, few modern concepts
    - Con: Shrinking user base
    - Con: Composability isn't great
    - Con: Little support for generic programming, GPUs, ...
    - Do you *really* want to ...?
- Scala, Go, Kotlin etc.:
    - Pro: Lots of individual strengths
    - Con: Math either fast *or* generic *or* or complicated
    - Con: Calling C, Fortran or Phython code often difficult
    - Con: Composability isn't great

The 97 and the 3 Percent

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

Donald E. Knuth

- Some programming languages (e.g. Python) great for the 97% - but can't make the 3% fast.
- Some other languages (e.g. C/C++, Fortran) can handle the 3% - but makes the 97% complicated.

The Two-language Problem

- Common approach nowadays:
  Write time critical code in C/C++, rest in Python
- Pro: End-user can code comfortably in Python, with good performance
- Con: Complexity of C/C++ **plus** complexity of Python
- Con: Need proficiency in **two** languages, barrier that prevents non-expert users from contributing to important parts of code
- Con: Limits generic implementation of algorithms
- Con: Severely limits metaprogramming, automatic differentiation, etc.

The Expression Problem

> *The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).*

Philip Wadler

- In other words: The capability to add both new subtypes and new functionality for a type defined in a package you don't own
- Object oriented languages typically can't do this
  (Ruby has a dirty way, Scala a clean workaround)
- If you have programming experience, you have felt this, even if you didn't name it
- Result: Packages tend not to compose well

We'd like to have a language ...

- as fast as C/C++/Fortran
- as easy to learn and productive as Python
- with a solution for the expression problem
- with first class math support (vectors, matrices, etc.)
- with true functional programming
- with great Fortran/C/C++/Python integration
- with true metaprogramming (like Lisp or Scala)
- good at parallel and distributed programming
- suitable for for interactive, small and large applications

Julia

- Designed for scientific/technical computing
- Originated at MIT, first public version 2012
- Covers the whole wish-list
- Clear focus on user productivity and software quality
- Rapid growth of user base and software packages
- Current version: Julia v1.6 (v1.7 will be out soon)

Julia Language Properties

- Fast: JAOT compilation to native CPU and GPU code
- Multiple-dispatch (more powerful than object-oriented): solves the expression problem
- Dynamically typed
- Very powerful type system, types are first-class values
- Functional programming and metaprogramming
- First-class math support (like Fortran or Matlab)
- ...

Julia Language Properties, cont.

- ...
- Local and distributed code execution
- State-of-the-art multi-threading: parallel code
  can call parallel code that can call parallel code, ...,
  without oversubscribing threads
- Software package management:
  Trivial to create and install packages
- Excellent REPL (console)
- Easy to call Fortran, C/C++ and Python code

Julia large-scale use case examples

- Celeste: Variational Bayesian inference for astronomical images (doi:10.1214/19-AOAS1258), 1.54 petaflops using 1.3 million threads on 9,300 Knights Landing (KNL) nodes on Cori at NERSC
- Clima: Full-earth climate simulation, https://clima.caltech.edu, large team, uses everything from MPI to GPUs
- ...

When (not) to use Julia

- *Do* use Julia for computations, visualization, data processing ... pretty much anything scientific/technical
- *Do not* use Julia for scripts what will only run for a second (code gen overhead), use Python or shell scripts
- *Do not* use Julia for non-computing web apps, etc. (*at least not yet*), use Go or Node.js

C/C++/Fortran-level speed

In [4]:

```
function mysum(A)
    s::eltype(A) = 0
    @inbounds @simd for x in A; s += x; end
    return s
end

@code_native debuginfo=:none mysum(rand(10^5))
```

```
        .text
        movq    8(%rdi), %rax
        testq   %rax, %rax
        je      L26
        movq    (%rdi), %rcx
        cmpq    $16, %rax
        jae     L31
        vxorpd  %xmm0, %xmm0, %xmm0
        xorl    %edx, %edx
        jmp     L144
L26:
        vxorps  %xmm0, %xmm0, %xmm0
        retq
L31:
        movq    %rax, %rdx
        andq    $-16, %rdx
        vxorpd  %xmm0, %xmm0, %xmm0
        xorl    %esi, %esi
        vxorpd  %xmm1, %xmm1, %xmm1
        vxorpd  %xmm2, %xmm2, %xmm2
        vxorpd  %xmm3, %xmm3, %xmm3
        nopl    (%rax,%rax)
```

```
L64:
        vaddpd  (%rcx,%rsi,8), %ymm0, %ymm0
        vaddpd  32(%rcx,%rsi,8), %ymm1, %ymm1
        vaddpd  64(%rcx,%rsi,8), %ymm2, %ymm2
        vaddpd  96(%rcx,%rsi,8), %ymm3, %ymm3
        addq    $16, %rsi
        cmpq    %rsi, %rdx
        jne     L64
        vaddpd  %ymm0, %ymm1, %ymm0
        vaddpd  %ymm0, %ymm2, %ymm0
        vaddpd  %ymm0, %ymm3, %ymm0
        vextractf128    $1, %ymm0, %xmm1
        vaddpd  %xmm1, %xmm0, %xmm0
        vpermilpd       $1, %xmm0, %xmm1        # xmm1 = xmm0[1,0]
        vaddsd  %xmm1, %xmm0, %xmm0
        cmpq    %rdx, %rax
        je      L157
        nopw    %cs:(%rax,%rax)
L144:
        vaddsd  (%rcx,%rdx,8), %xmm0, %xmm0
        incq    %rdx
        cmpq    %rdx, %rax
        jne     L144
L157:
        vzeroupper
        retq
        nopw    %cs:(%rax,%rax)
```

Packages compose due to multiple dispatch

In [8]:
```
using Unitful
a = 5.0u"m/s"
```

Out[8]:
```
5.0 m s^-1
```

In [7]:
```
sizeof(a)
```

Out[7]:
```
8
```

In [10]:
```
using Measurements
sqrt(40 ± 2)
```

Out[10]: 6.32 ± 0.16

`Unitful` and `Measurements` do "don't know" each other (no dependency between them), and yet:

In [11]:
```
sqrt(5.0u"m/s" ± 0.1u"m/s")
```

Out[11]:
```
2.236 ± 0.022 m^1/2 s^-1/2
```

Types are first-class values

Can pass types are function arguments, can calculate with types (without runtime reflection!):

In [16]:
```
function precise_sum(A)
    R = widen(eltype(A))
    s::R = 0
    @inbounds @simd for x in A; s += x; end
    return s
end

typeof(precise_sum(randn(Float32, 10^6)))
```

Out[16]:
```
Float64
```

Type hierarchy extends down to primitive types

Enables easy generic programming:

In [19]:
```
Float64 <: AbstractFloat <: Real <: Number <: Any
```

Out[19]:
```
true
```

Quite hard to express in other languages:

In [21]:
```
using LinearAlgebra
foo(x::Real) = x^2
foo(A::AbstractArray{<:Real}) = foo(det(A))

foo(42), foo(rand(5, 5))
```

Out[21]:
```
(1764, 9.534449531975872e-5)
```

Julia can glue languages together

Zero-overhead C and Fortran calls, inline Python, R, Matlab, C++ (Julia v1.3)

In [28]:

```
using PyCall

np = pyimport_conda("numpy", "numpy")

typeof(np.random.rand(100))
```

Out[28]:

```
Vector{Float64} (alias for Array{Float64, 1})
```

In [35]:

```
A = rand(10^6)
r = py"""
import numpy as np
print(np.sum($A))
"""
```

Automatic Differentiation

In [42]:

```
struct DenseLayer{M<:AbstractMatrix{<:Real},V<:AbstractVector{<:Real},F<:Function} <: Functi
    A::M
    b::V
    f::F
end

(l::DenseLayer)(x::AbstractVector{<:Real}) = (l.f).(l.A * x + l.b)

f_loss(y) = sum(y .^ 2);

mylayer = DenseLayer(rand(5,5), rand(5), x -> ifelse(x > zero(x), x, zero(x)))
x = rand(5)

using Zygote
g = Zygote.gradient((mylayer, x) -> f_loss(mylayer(x)), mylayer, x)
g[1].A
```

Out[42]:

```
5×5 Matrix{Float64}:
 2.16695  0.916992  3.32997  1.91845  2.21835
 2.44831  1.03606   3.76234  2.16755  2.50639
 3.13557  1.32688   4.81845  2.77599  3.20994
 1.95584  0.827654  3.00555  1.73155  2.00223
 1.04714  0.443121  1.60915  0.92706  1.07198
```

GPU Computing

Julia code can compile natively to GPUs:

In [43]:
```
using CUDA
```

In [47]:
```
mylayer = DenseLayer(cu(rand(5,5)), cu(rand(5)), x -> ifelse(x > zero(x), x, zero(x)))
x = cu(rand(5))
Zygote.gradient((mylayer, x) -> f_loss(mylayer(x)), mylayer, x)[1].A
```

Out[47]:
```
5×5 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 0.120834   1.64269   2.22928   1.02623   0.879836
 0.102      1.38666   1.88183   0.866279  0.742705
 0.0942255  1.28096   1.73838   0.800247  0.686092
 0.156132   2.12255   2.88051   1.32601   1.13686
 0.0504331  0.685619  0.930451  0.428323  0.367223
```

Last but not least

- Awesome package management with built-in reproducability (unique)
- User-friendly introspection and perf-opt tooling
- Welcoming, talented and science-heavy community
- no BDFL

Final Remarks

- Julia is productive, fast and fun - give it a chance!
- Multiple dispatch opens up powerful ways of combining code
- Awesome support for generic programming, heterogenous computing, automatic differentiation, ...