The Julia language: easy, generic, fast — but no free lunch

Henri Laurie

University of Cape Town

September 2022

Greeting

Welcome to this series of four seminars

Greeting

Welcome to this series of four seminars

THE TOPICS

Greeting

Welcome to this series of four seminars

THE TOPICS

1. Showcasing Julia: pride, charm, resources, gotchas

Greeting

Welcome to this series of four seminars

THE TOPICS

1. Showcasing Julia: pride, charm, resources, gotchas
2. The why and how: design principles and tradeoffs

Greeting

Welcome to this series of four seminars

THE TOPICS

1. Showcasing Julia: pride, charm, resources, gotchas
2. The why and how: design principles and tradeoffs
3. `DifferentialEquations.jl`: best in class

Greeting

Welcome to this series of four seminars

THE TOPICS

1. Showcasing Julia: pride, charm, resources, gotchas
2. The why and how: design principles and tradeoffs
3. `DifferentialEquations.jl`: best in class
4. Solving the two-language problem and the expression problem: Julia for HPC?

Outline

Why was Julia created?

Why was Julia created?

Public (and punchy, and charming) answers to this question:

Why was Julia created?

Public (and punchy, and charming) answers to this question:

- the first public notice: `https://julialang.org/blog/2012/02/why-we-created-julia/`

Why was Julia created?

Public (and punchy, and charming) answers to this question:

- the first public notice: `https://julialang.org/blog/2012/02/why-we-created-julia/`
- a talk of similar vintage; "Rationale Behind Julia and the Vision" `https://www.youtube.com/watch?v=02U9AJMEWx0`

Why was Julia created?

Public (and punchy, and charming) answers to this question:

- the first public notice: `https://julialang.org/blog/2012/02/why-we-created-julia/`
- a talk of similar vintage; "Rationale Behind Julia and the Vision" `https://www.youtube.com/watch?v=02U9AJMEWx0`
- an early(-ish) paper: Bezanson *et al.*, arXiv 19 July 2015

Why was Julia created?

Public (and punchy, and charming) answers to this question:

- the first public notice: `https://julialang.org/blog/2012/02/why-we-created-julia/`
- a talk of similar vintage; "Rationale Behind Julia and the Vision" `https://www.youtube.com/watch?v=02U9AJMEWx0`
- an early(-ish) paper: Bezanson *et al.*, arXiv 19 July 2015
- and a recent podcast: `https://www.youtube.com/watch?v=W6I1zQ16_44`

Interactive dynamism PLUS high performance

. . . but the title of this slide is the answer.

Interactive dynamism PLUS high performance

. . . but the title of this slide is the answer.

The four founders (Bezanson, Edelman, Karpinski, Shah) had different but compatible motivation: they were very pleased to do a lot of work in interactive environments like Matlab, Python and R, but also very much appreciated the speed and fine-grained control available in C and C++.

Interactive dynamism PLUS high performance

. . . but the title of this slide is the answer.

The four founders (Bezanson, Edelman, Karpinski, Shah) had different but compatible motivation: they were very pleased to do a lot of work in interactive environments like Matlab, Python and R, but also very much appreciated the speed and fine-grained control available in C and C++.

The introductory paragraphs of the launching blog post makes the motivation pretty clear!

The old way: glue together code from many languages

The old way: glue together code from many languages

This is the two-language problem and was seriously proposed as best practice by Ousterhout in 1998.

The old way: glue together code from many languages

This is the two-language problem and was seriously proposed as best practice by Ousterhout in 1998.

"A fundamental change is occurring in the way people write computer programs, away from system programming languages such as C or C++ to scripting languages such as Perl or Tcl. . . . scripting languages will handle many of the programming tasks in the next century better than system programming languages. System programming languages were designed for building data structures and algorithms from scratch, starting from the most primitive computer elements. Scripting languages are designed for gluing. They assume the existence of a set of powerful components and are intended primarily for connecting components."

The old way: glue together code from many languages

This is the two-language problem and was seriously proposed as best practice by Ousterhout in 1998. (Ousterhout, J.K. (1998) Scripting: higher level programming for the 21st Century. Computer 31(3) 23–30.)

"A fundamental change is occurring in the way people write computer programs, away from system programming languages such as C or C++ to scripting languages such as Perl or Tcl. . . . scripting languages will handle many of the programming tasks in the next century better than system programming languages. System programming languages were designed for building data structures and algorithms from scratch, starting from the most primitive computer elements. Scripting languages are designed for gluing. They assume the existence of a set of powerful components and are intended primarily for connecting components."

The Julian way: write the whole project in Julia

The Julian way: write the whole project in Julia

Doesn't quite happen in practice, of course . . .

The Julian way: write the whole project in Julia

Doesn't quite happen in practice, of course . . .

eg `https://github.com/JuliaLang/julia` shows significant contribution from other languages—but a remarkable $\approx 70\%$ is written in Julia itself.

The Julian way: write the whole project in Julia

Doesn't quite happen in practice, of course . . .

eg `https://github.com/JuliaLang/julia` shows significant contribution from other languages—but a remarkable $\approx$ 70% is written in Julia itself.

This percentage is even higher in some of the big projects that use Julia—eg `DifferentialEquations.jl`, the topic on 21 September, is 100% Julia.

High speed

High speed

Petaflop performance: the Celeste project

`https://juliacomputing.com/case-studies/celeste/`

High speed

Petaflop performance: the Celeste project
`https://juliacomputing.com/case-studies/celeste/`

Matches C and Fortran in micro-benchmarks:
`https://julialang.org/benchmarks/`

High speed

Petaflop performance: the Celeste project
`https://juliacomputing.com/case-studies/celeste/`

Matches C and Fortran in micro-benchmarks:
`https://julialang.org/benchmarks/`

Of course, speed as a generalised attribute of a language is hard to measure.

High speed

Petaflop performance: the Celeste project
`https://juliacomputing.com/case-studies/celeste/`

Matches C and Fortran in micro-benchmarks:
`https://julialang.org/benchmarks/`

Of course, speed as a generalised attribute of a language is hard to measure. Speed as an experience possibly even harder to measure, but of course dominates conversations, blogs and comments!

High speed

Petaflop performance: the Celeste project
`https://juliacomputing.com/case-studies/celeste/`

Matches C and Fortran in micro-benchmarks:
`https://julialang.org/benchmarks/`

Of course, speed as a generalised attribute of a language is hard to measure. Speed as an experience possibly even harder to measure, but of course dominates conversations, blogs and comments!

For me personally, I came for the speed, and stayed also for the speed and have not had any real disappointments.

Recognition

The 2019 James H. Wilkinson prize for Numerical Software went to Bezanson, Karpinski and Shah (Edelman didn't qualify). From the SIAM News announcement: "Julia allows researchers to write high-level code in an intuitive syntax and produce code with the speed of production programming languages."

Recognition

The 2019 James H. Wilkinson prize for Numerical Software went to Bezanson, Karpinski and Shah (Edelman didn't qualify). From the SIAM News announcement: "Julia allows researchers to write high-level code in an intuitive syntax and produce code with the speed of production programming languages."

Rankings: not quite in the top 20 but near there (several rankings).

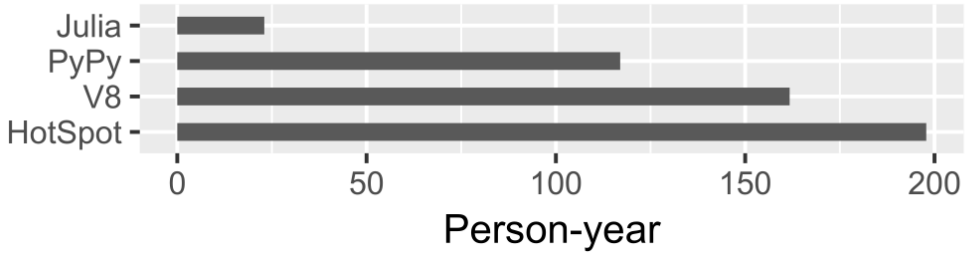
Recognition

The 2019 James H. Wilkinson prize for Numerical Software went to Bezanson, Karpinski and Shah (Edelman didn't qualify). From the SIAM News announcement: "Julia allows researchers to write high-level code in an intuitive syntax and produce code with the speed of production programming languages."

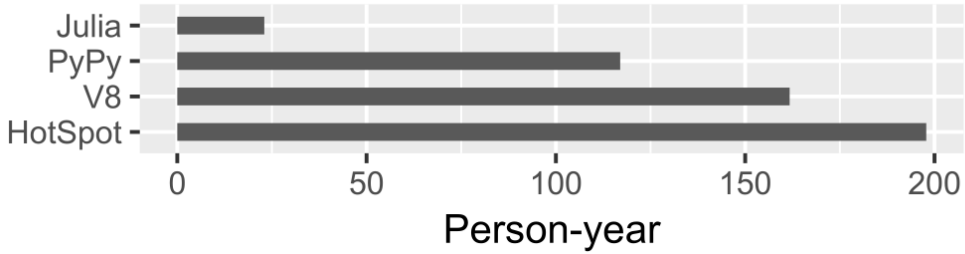
Rankings: not quite in the top 20 but near there (several rankings).

Notable projects: Celeste, CliMA (for more, see Julia Computing's home page, though it is rather biased).

Rapid development



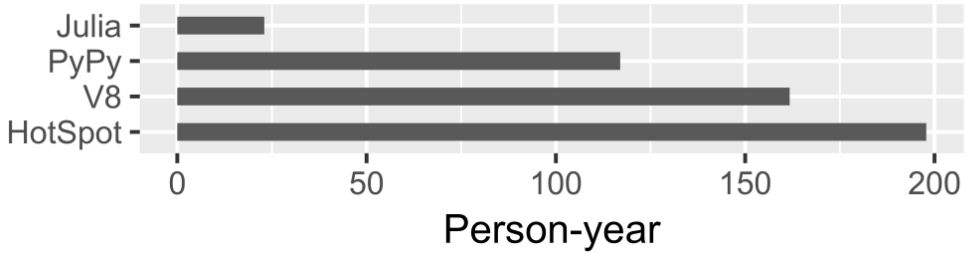
Rapid development



"... shows the person-years invested in several language implementations. These rough measures were obtained using commit histories: two commits made by the same developer in one week were counted as one person-week of effort."

Rapid development

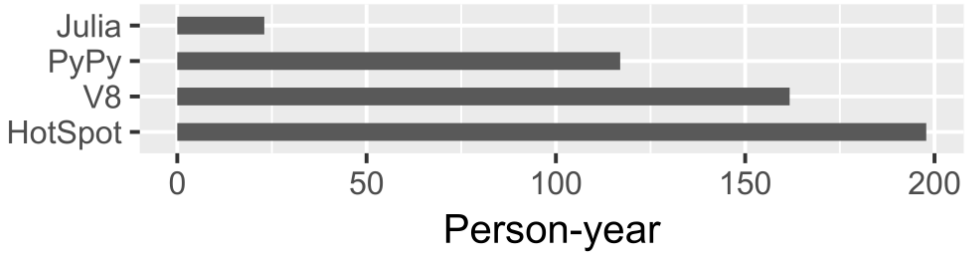
From Bezanson *et al.* (Proc. ACM Program. Lang. 2019) Julia: dynamism and performance reconciled by design.



"... shows the person-years invested in several language implementations. These rough measures were obtained using commit histories: two commits made by the same developer in one week were counted as one person-week of effort."

Rapid development

From Bezanson *et al.* (Proc. ACM Program. Lang. 2019) Julia: dynamism and performance reconciled by design.



"... shows the person-years invested in several language implementations. These rough measures were obtained using commit histories: two commits made by the same developer in one week were counted as one person-week of effort."

`DifferentialEquations.jl` (21 Sep talk) also shows remarkable output for low investment of programming effort.

Julia code is a pleasure to write and to read

(*examples later*)

Julia code is a pleasure to write and to read

(*examples later*)

Julia takes lessons from Python and Matlab; my subjective experience is that it improves on both.

Julia code is a pleasure to write and to read

(*examples later*)

Julia takes lessons from Python and Matlab; my subjective experience is that it improves on both.

My view is that the ease of coding and code reading in Julia goes far beyond these lessons: it comes from multiple dispatch (on which more later).

Overview

Straightforward, elegant, interactive, performant, adaptable, composable, extendible

The REPL

Provides interactivity. Evaluates every logically complete Julia line entered.

The REPL

Provides interactivity. Evaluates every logically complete Julia line entered.

(*demo*)

The REPL

Provides interactivity. Evaluates every logically complete Julia line entered.

(*demo*)

Can run code files (they're plain text with `.jl` file extension).

The REPL

Provides interactivity. Evaluates every logically complete Julia line entered.

(*demo*)

Can run code files (they're plain text with `.jl` file extension).

Has three further modes: help, shell and Pkg.

Unicode!

So easily available!

Unicode!

So easily available! Here are the available unicode symbols

`https://docs.julialang.org/en/v1/manual/unicode-input/`

Unicode!

So easily available! Here are the available unicode symbols
`https://docs.julialang.org/en/v1/manual/unicode-input/`

Really makes formulae look like maths:
`https://discourse.julialang.org/t/seven-lines-of-julia-examples-sought/50416/132`

Unicode!

So easily available! Here are the available unicode symbols
`https://docs.julialang.org/en/v1/manual/unicode-input/`

Really makes formulae look like maths:
`https://discourse.julialang.org/t/seven-lines-of-julia-examples-sought/50416/132`

Let's try the following out in the REPL

$\alpha^2(x) = x^2; \alpha^3(x) = x^3$

$\alpha = 22$
$\sqrt{\alpha^3(\alpha)} == 22^{3/2}$ `# type-insensitive equality comparison`

🐈 = 5

2🐈

Example of user-extension: `OhMyREPL`

`https://kristofferc.github.io/OhMyREPL.jl/latest/`

Let's try it out

Code introspection!

Some like to say it like this: Julia is homoiconic

Well illustrated by the macro `@code_llvm`.

Multiple dispatch

Crucial design decision (more on 14 Sep):

Multiple dispatch

Crucial design decision (more on 14 Sep):

generic functions with multimethods are the default,

the method to execute is selected by type signature (i.e. types of every field in call signature),

method selection happens at runtime,

Multiple dispatch

Crucial design decision (more on 14 Sep):

generic functions with multimethods are the default,

the method to execute is selected by type signature (i.e. types of every field in call signature),

method selection happens at runtime,

from OOP point of view, this prefers composition to inheritance (and seems to be superior; I'll return to this on 28 Sep),

Multiple dispatch

Crucial design decision (more on 14 Sep):

generic functions with multimethods are the default,

the method to execute is selected by type signature (i.e. types of every field in call signature),

method selection happens at runtime,

from OOP point of view, this prefers composition to inheritance (and seems to be superior; I'll return to this on 28 Sep),

user-defined types for purpose of calling user-defined code bodies is absolutely the standard paradigm in Julia packages.

Massively composable

A good example: `Unitful.jl`, `DifferentialEquations.jl` and `Plots.jl` are three independent packages.

Massively composable

A good example: `Unitful.jl`, `DifferentialEquations.jl` and `Plots.jl` are three independent packages.

`Unitful` extends mathematical operators so they work on dimensional values. Because of Julia's type system and multiple dispatch, this is enough to make the solvers of `DifferentialEquations` work for the dimensional values.

Massively composable

A good example: `Unitful.jl`, `DifferentialEquations.jl` and `Plots.jl` are three independent packages.

`Unitful` extends mathematical operators so they work on dimensional values. Because of Julia's type system and multiple dispatch, this is enough to make the solvers of `DifferentialEquations` work for the dimensional values.

`Plots` and `DifferentialEquations` already interact very well.

Massively composable

A good example: `Unitful.jl`, `DifferentialEquations.jl` and `Plots.jl` are three independent packages.

`Unitful` extends mathematical operators so they work on dimensional values. Because of Julia's type system and multiple dispatch, this is enough to make the solvers of `DifferentialEquations` work for the dimensional values.

`Plots` and `DifferentialEquations` already interact very well.

To make `Plots` work with dimensional values requires an extension: `UnitfulRecipes.jl`, which defines several new functions and three new types.

Massively composable

A good example: `Unitful.jl`, `DifferentialEquations.jl` and `Plots.jl` are three independent packages.

`Unitful` extends mathematical operators so they work on dimensional values. Because of Julia's type system and multiple dispatch, this is enough to make the solvers of `DifferentialEquations` work for the dimensional values.

`Plots` and `DifferentialEquations` already interact very well.

To make `Plots` work with dimensional values requires an extension: `UnitfulRecipes.jl`, which defines several new functions and three new types. Then the three-way composition also works!

The virtues of Julia package management

It is well integrated with the interactive environments.

It is supported by a worldwide network of servers.

It enables project management via git (fully integrated).

It can guarantee fully reproducible environments.

A few other interactive environments

IJulia (Jupyter notebooks).

Pluto (reactive, not merely interactive).

The VS Code (Codium) extension for Julia.

Brief demo of IJulia and Pluto

Julia's people

Very approachable, very responsive and friendly community (warmer and more respectful than most in my experience!)

But the communication is somewhat inhouse: more Julia Discourse, Zulip and Github than Stackexchange and Reddit.

As noted, the package system is superb (even offers to install missing packages, and does so with no fuss at all if asked).

The two major public faces are the Julia Language group (`https://julialang.org`) and the Julia Computing company (`https://juliacomputing.com`).

Compilation costs

Have to be paid!

But most of time they are not onerous. Load times used to be long but despite Julia's growth they are markedly lower now.

However, recompilations can be a pain: the TTFP and related phenomena. This situation is likewise improving (but will never entirely disappear).

When high performance stays away ...

Does happen . . . sigh . . .

But AFAIK has always yielded to concerted effort—that is, Julia does not seem to have any areas of necessarily low performance.

Error messages (oh dear)

Can be opaque, even very opaque, eg. they might reference `Union` types that take up dozens of lines.

Also, they are almost entirely stack traces. Very useful if you know how to read them, very daunting at first.

Very large runtime

This makes distributing small free-standing programs, apps difficult.

That said, static compilation and other developments will soon make this far less of a problem.

Julia's tooling a tad skimpy

The interactive offerings work well, as we saw.

Profiling, debugging, editing: it's there, it works but not as well as for, say, Python and C++.

As to autogenerated code . . . I don't know (but does it matter?).

Integration with git is excellent, therefore so is project management, thanks to `Pkg`.

Large projects

Definitely possible (Celeste in pre-v1 days, CliMA now, DifferentialEquations.jl pre- and post-v1)

. . . but I've seen discussions that this is largely up to the group concerned, its policies and leadership, and less because Julia has good ways of ensuring effective collaboration.

Closing remarks

So this is Julia: easy, generic and fast.
But certainly no free lunch!

Looking forward to the next three presentations:

14 Sep goes into the technical aspects of Julia
21 Sep presents DifferentialEquations.jl,
the best-in-class general-purpose package for DEs
28 Sep returns to the expression and two-language problems,
and in that light discusses Julia for HPC

THANKS!