Nathan Robinson

COSC 4318, ALC

Dr. Smith

April 16, 2018

Juila: The Best of Both Worlds

**Rationale for Julia**

Computing science is in an era of large scale data analytics. There is access to a massive amount of information from nearly every major industry. There have been many tools and application built to process, organize, and interpret the vast data available, but progress has been painfully slow, especially for systems that require complex evaluation. These half-measures have struggle to keep up with the pace of big data due to the lack of an adequate language platform. One of major goals for Julia is to fill that space. Julia is “a high-level, high-performance dynamic programming language for numerical computing” [1]. It offers a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library.

The core creators of Julia were Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman. They claimed the existing technical computing tools were insufficient for the current demands of the field. Prototyping requires a “high-level, easy-to-use, and flexible language that lets the developer concentrate on the problem itself instead of on low-level details of the language and computation” [2]. Technical programmers had to use old, interpreted languages like Matlab, R, or Python to express the problem at a high level. However, the computation of that problem requires the fastest possible performance, so the solution has to be rewritten in statically compiled languages like C or Fortran. Julia was designed to meet both needs, providing developer-friendly and expressive vocabulary and syntax, and offering competitive execution speed by using CPU and memory resources as efficiently as C.

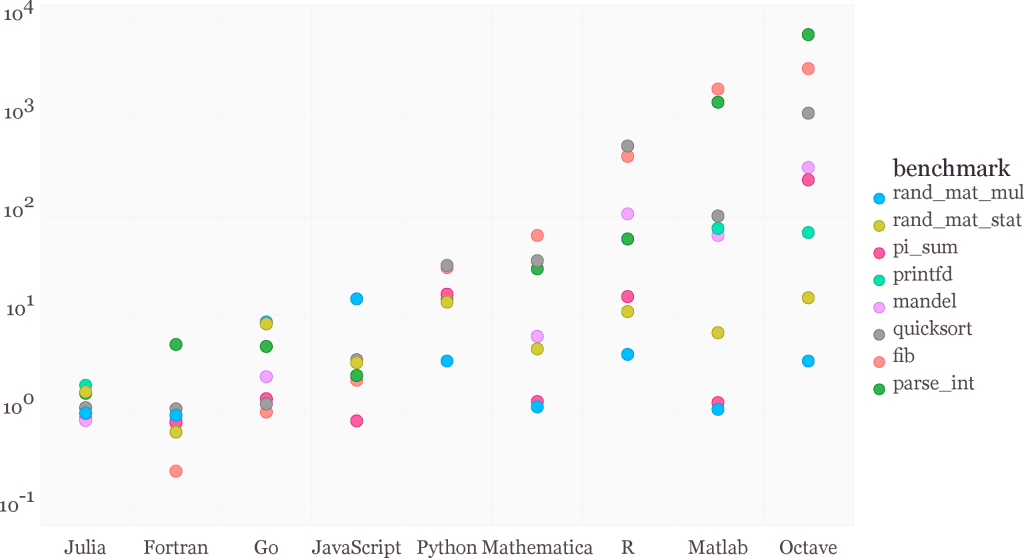


Figure 1: Benchmark times relative to C (smaller is better, C performance = 1.0) [1]

Julia has two primary technical computing predecessors. The first is Matlab, created by MathWorks in 1984. Matlab, or Matrix Laboratory, was designed to model matrices and other complex mathematical system. It became very popular for educational purposes in university math and engineering environments. The second is Python, or more specifically, NumPy. Python was created by Guido van Rossum in 1991, and Numeric (later NumPy) was introduced in 1995. Python has reigned as the go-to language for mathematical processing, thanks in large to its large math function libraries and open community development. Neither of these languages are designed to offer the execution speeds of a statically compiled language, however, Julia is:



Figure 2: Comparative Performance of JuMP, a Numerical Optimization Package Written in Julia, Relative to AMPL, a Commercial Solver (Source: Miles Lubin and Iain Dunning.) [3]

By reducing the need to rewrite code in a low-level language, “Julia developers have proven that working in one environment that has the expressive capabilities as well as the pure speed is possible using the recent advances in Low Level Virtual Machine Just in Time (LLVM JIT) compiler technologies” [2].

**Specification in Summary**

* Computational power and speed without leaving the Julia environment
* Dynamic type system: types for documentation, optimization, and dispatch
  + User-defined types are as fast and compact as built-ins
* Multiple dispatch: define function behavior across many combinations of argument types
  + Automatic generation of efficient, specialized code for different argument types
  + Elegant and extensible conversions and promotions for numeric and other types
* High performance: near statically-compiled languages
* Homoiconicity: macros and metaprogramming abilities to increase abstraction power
* Parallelism: built-in concurrent and parallel tools to allow distributed computation
* Powerful shell-like capabilities for managing other processes
  + Coroutines: lightweight “green” threading
  + Thrives in the multicore environment
* Interop: Can call Python, C, and Matlab library functions, and vice virsa
  + Efficient support for Unicode, including but not limited to UTF-8
* Free and open source with a liberal MIT license
* General purpose: clear, dynamic, and interactive syntax that is easy to use and learn
  + Looks like the pseudo code with an obvious and familiar mathematical notation
* Built-in package manager

**Foundation of Julia**

Julia unites the dynamic, untyped, and interpreted languages (Python, Ruby, Perl, MATLAB/Octave, R…) and statically typed and compiled languages (C, C++, Fortran, and Fortress). Julia does this by removing the static compilation step, allowing for flexibility and speed. A LLVM-based JIT compiler generates the machine code. The design of the language uses this compiler to achieve high performance for numerical, technical, and scientific computing.

**Type System**

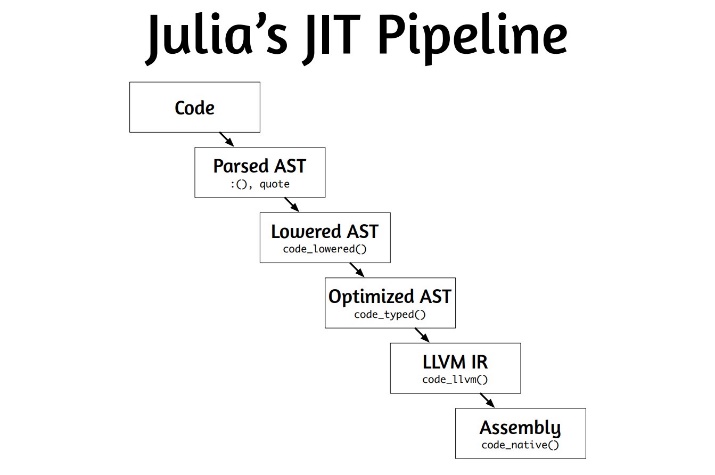
Julia’s type system is dynamic, which enables its high-performance benchmarks. Type information is collected by an automated type-inference engine that interprets the type from the data in the variables. There is no compile-time type, meaning variable do not have types, only values. Secondly, the type system is nominative: the relationships among types are explicitly declared. Similar to Dart, declaring the types of variable is optional, but can be useful for documentation, tooling abilities, and helping the compiler optimize the execution path. Also, concrete types are final. Lastly, the system is parametric. Abstract and concrete types can be parameterized by other types and certain values. Typeless Julia is like traditional dynamic languages, but it still runs at statically compiled speeds. [3]

**Multiple Dispatch**

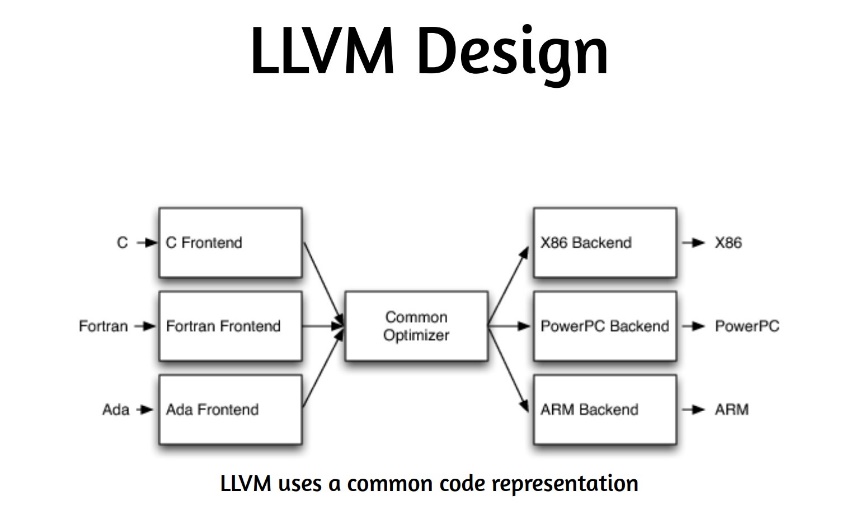
Julia is built around generic programming and polymorphism. Class objects can be easily passed vertically, extended, and implemented. All functions generic, are thus extremely flexible across different types. When a function is called, its definition depends on the types of all its arguments. For example, size is a generic function with fifty concrete method implementations. This is where the dynamic multiple dispatch system uses all of a function’s arguments to efficiently pick the optimal method from several method definitions. Depending on the types and data-flow inferences, specific native code implementations of the function are generated to maximize efficiency. User-defined functions can be easily overloaded for any combination of argument types. Its type system aligns closer with primitive machine operations. Exhaustive testing is compulsory because types are not statically checked, so type errors can occur at run-time.

**Speed**

One of the primary goals for Julia is speed. The benchmark comparison with other languages on page 1 demonstrates that Julia rivals C and FORTRAN. It is much faster than traditional dynamic languages and consistently is within a factor of two of fully optimized C code. With speeds like this, dropping down to a low-level language is unnecessary. Libraries could be developed in a high-level language instead of in C or FORTRAN. Julia compiles code at run-time, otherwise know as just-in-time compilation, or JIT, like Java. Source code is parsed into an abstract syntax tree which is then lowered and optimized. The last step before assembly is to use a LLVM to further optimize and generate code. Then each method is translated into machine code. [4]



Notice the absence of a static compilation step. The LLVM, or low level virtual machine, takes care of the code generation without the need of static compilation. LLVM was introduced by Vikram Adve and Chris Lattner at University of Illinois in 2003. It is a collection of modular compiler and toolchain technologies. This system allows Julia program execution time to be competitive with that of C and Fortran.



**Features**

Julia embodies multiple paradigms. It bares facets of procedural, functional, metaprogramming, and object oriented languages. It is not class-based like Java, Ruby, or C#, but its type system can be used as a powerful kind of inheritance. User-defined types are as efficient as built-in types and conversions and promotions for all types are quick. Functions are first-class objects in Julia, so designing programs with pure functions has no side effects.

Metaprogramming facilities are a structural part of Julia. The language inherited homoiconic properties from Lisp. Homoiconicity means that code lives in data structures that can be manipulated by the language itself. Users can create Lisp-like macros to directly manipulate expressions, or use can use ‘Expr’ and ‘Symbol’ types to build powerful abstraction tools.

Parallel computing is Julia is available through distributed memory using multiple processes on one or more nodes, through shared memory using multiple processes on a single node, and through multi-threading. The @parallel and pmap functions break up computations across processors and machines. Distributed arrays can be used to split large matrices across programmers, and @sync primitives can push data back and forth across machines. This message passing model enables programs to run via multiple local and remote processes.

This Julia example “demonstrates how to count the number of heads in a large number of coin tosses in parallel using @parallel:

nheads = @parallel (+) for i = 1:100000000

rand(Bool)

end

The computation is automatically distributed across all available compute nodes, and the result, reduced by summation (+), is returned at the calling node” [1].

Julia offers strong support for community development. Cluster computing enables coders to work without being on the local machine. To interact with Julia a few environments are available. Most notable are REPL, Gadfly, and IJulia notebook. IJulia notebooks are the primary way the language is used currently. It make cloud-based computing simple, as well as enabling easy sharing, group editing, debugging, visualization, analysis, and data management. REPL stands for real-evaluation-print loop, an environment like Python or MATLAB. Gadfly is a visualization tool that makes plotting for MATLAB, R, and almost any program, very easy.



Figure 3: Web-based interactive IJulia Notebook session, using Gadfly. JuliaBox provides a way to run IJulia notebooks in your browser on Docker sandboxed containers provisioned on demand [1]

**Comparing Similar Languages**

Julia excels at interoperation with similar languages. There are packages to call several languages into the Julia environment. MATLAB syntax is already very close to Julia syntax, and has the same matrix computational power, but is free and open license. The benchmarks indicate Julia is 10 to 1,000 faster for certain operations. Julia provides an interface to the MATLAB language with the package MATLAB.jl. Julia is also the natural successor to R, the go-to language for statistical operations. Julia is just as usable but is again 10 to 1,000 times faster. Julia has a richer type system than the vector-based types of R. Julia provides an interface to the R language with the package Rif.jl (<https://github.com/lgautier/Rif.jl>). Statistics in MATLAB is frustrating, as is linear algebra in R, but both are easy with Julia.

Python is 10 to 30 times slower than Julia. Julia also compiles the code that reads like Python into machine code that performs like C. Python functions can be called from within Julia using the PyCall package (https://github.com/stevengj/PyCall.jl). PyCall.jl offers automatic conversion of types between Julia and Python. Julia can be called by C, C++, Python, and other language using custom packages. Like Perl, Julia has a modern Unicode capable string processing and regular expressions. It can also be used as a glue language to synchronize the execution of other programs or to manage other processes at the shell level. Julia has a standard library written in Julia and a built-in package manager based on GitHub, called Metadata, to handle the budding collection of external libraries. It is cross platform, supporting GNU/Linux, Darwin/OS X, Windows, and FreeBSD for both x86/64 (64-bit) and x86 (32-bit) architectures.

Syntax in Julia is very similar to pseudocode, a mix between Python and MATLAB almost. This enables readability of the code and quick learning, essential aspects of any language meant for collaboration. All mainstream data structures (strings, sets, tuples, dictionaries, stacks, vectors, matrices) are supported by the standard library. Niche structures and functions are available through other language libraries and community packages.

Because of the huge number of existing libraries in all these languages, Julia can be easily mixed with R or Python when the needed. Julia shines when handling linear algebra algorithms and graph analysis techniques. These strengths make the language ideal for big data and predictive analysis. High Performance Computing (HPC) domain experts can use Julia to quickly express problems in such a way that they can easily use modern HPC hardware. [5]

**Community**

Julia is a relatively new language, but is gaining momentum in the scientific and technical computing domain quickly. As of 2014 it is being taught in courses at Stanford, Penn State, Cornell, MIT, Western and several smaller universities. The community is small but growing and active. Members are encouraged to make developments and share fixes. Everything is open-source. Participating in design discussions and making contributions is easy; everything goes through the ‘julialang’ repo on Github.

“As of the v0.2 point release of November 17, 2013, the Julia repository has logged 15,285 commits by 142 developers, who together have written 204,408 lines of code” [3]. In 2013 the Github repository had 2,741 stars. Now Julia has 41,590 commits, 13086 stars, and 686 contributors on Github [6]. There are many Julia development communities across the globe.

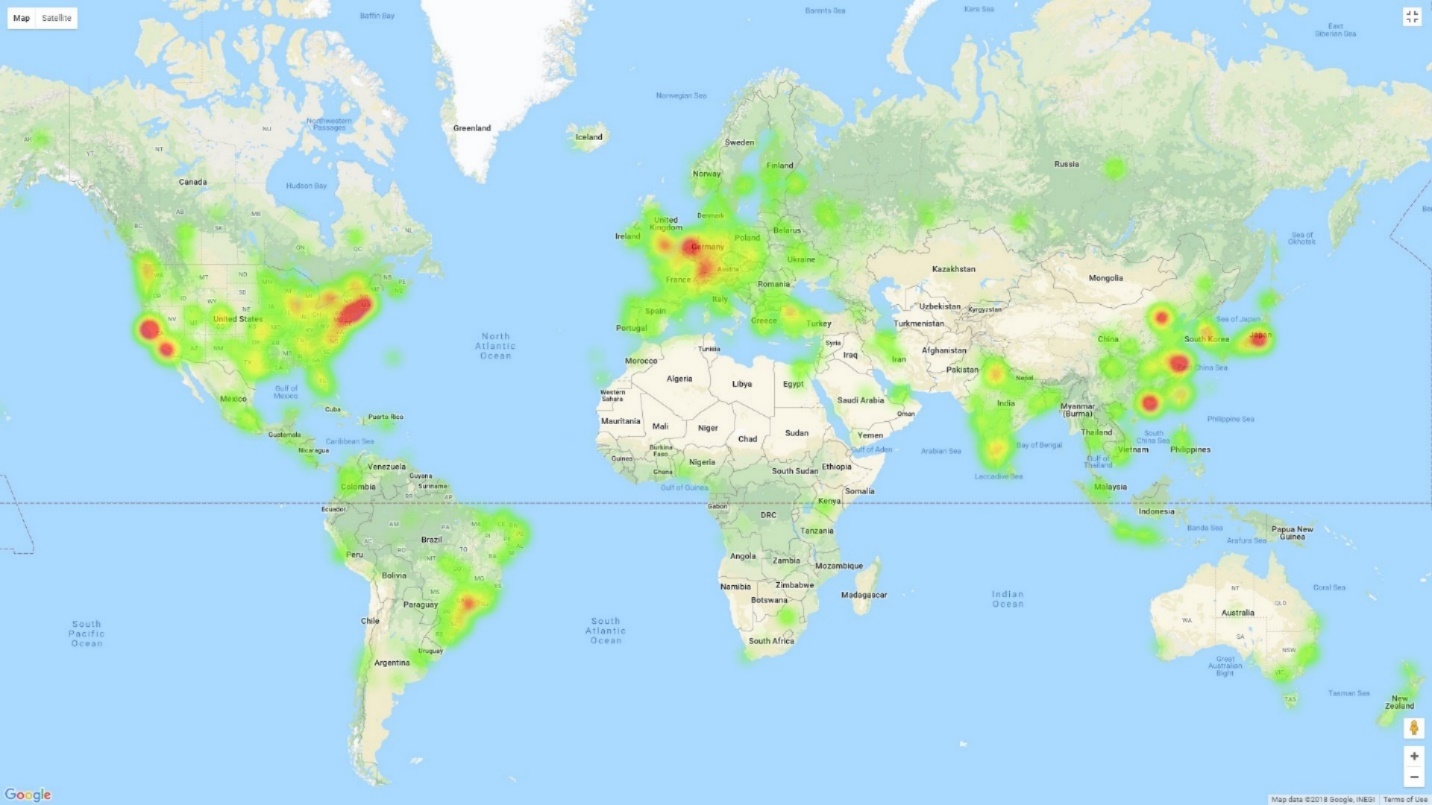


Figure 4: Heatmap visualization of Github repositories [7]

**Conclusion**

*Why would you use Julia?*

If you are a technical computing programmer, use this language. The main advantage is its ability to generate specialized code for different combinations of parameter types. Code can be written at an abstract level but still achieve the efficiency associated with the low-level languages because of the compiler’s ability to infer these types. Julia unites the best of both worlds into one environment, something which most researchers and language designers thought was impossible.

Julia is stable and production ready and the learning curve is gentle. It is easy to use productively, even if you do not use the fancy features. Prototyping can be done quickly without sacrificing execution speed, the type system makes expressing algorithms simple, it is easy to parallelize code, and it has an extensive standard library written in itself. Julia connects with other languages well, and the community is active and helpful.

Julia creates clear, concise code that is easy to implement and modify. When it is coded well it is very fast, and it can combine matrix operations with loop-based operations. Python, MATLAB, Java, R, C…none of those languages can fill all three of those roles.

*Why wouldn’t you use Julia?*

If you are a neural network researcher, not just a user, the GPU infrastructure is not yet complete. If you are writing production code, understand that interfaces may change because the language is still growing. If you goal is to make widely used packages use a more popular language. [8]

“We want a language that’s open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that’s homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled” -Creators of Julia [1].

# References

|  |  |
| --- | --- |
| [1] | MIT, "Home," [Online]. Available: https://julialang.org/. [Accessed 11 4 2018]. |
| [2] | I. Balbaert, "Getting Started with Julia Prograqmming," PACKT Publishing, NA, 2015. |
| [3] | J. Chen, "Open Big Data Computing with Julia," 10 12 2013. [Online]. Available: http://istc-bigdata.org/index.php/open-big-data-computing-with-julia/. [Accessed 12 4 2018]. |
| [4] | C. Kwon, Julia Programming for Operations Research: A Primer on Computing, PACKT Publishing, 2016. |
| [5] | A. Sengupta, Julia High Performance, 2016. |
| [6] | "The Julia Language: A fresh approach to technical computing.," [Online]. Available: https://github.com/JuliaLang/julia. [Accessed 11 4 2018]. |
| [7] | RedDrawf, "JuliaLang," RedDrawf, [Online]. Available: http://jrvis.com/red-dwarf/?user=JuliaLang&repo=julia. [Accessed 12 4 2018]. |
| [8] | M. Sherrington, Mastering Julia, PACKT Publishing, 2015. |