Julia - A new language for technical computing

Published in Automated Trader Magazine Issue 43 Q2 2017

Historically, developers have faced a trade-off between the ease of use of scripting languages and the power and flexibility of lower-level languages. Julia offers the best of both worlds - and allows easy interoperability with existing languages.

The programming language Julia started from a simple observation. Traditionally, programming languages that focused on numerical computing could be split into two distinct groups: a set of static languages (for example, C, C++, Fortran) that are fast for execution, but slow for development, and another set of dynamic languages (for example, Python, R, Matlab) that are often slow in their execution, but enable rapid development. The creators of Julia asked themselves if such a dichotomy was really necessary. Was there some sort of natural law preventing a numerical programming language from being both high-performing and productive? Overcoming the performance deficiencies of existing high-level scripting languages used in technical computing often requires programmers to choose between two options: either to re-implement portions of their code in low-level systems languages using interfaces provided by the high-level language, or to rewrite completely their high-level language application within a low-level language. To solve this 'two-language problem' a single language is required that addresses both the needs of the computer when constructing code that can be executed extremely fast, as well as the human desire for writing high-level expressive code that captures mathematical ideas in a concise and generic manner.

Julia provides a solution to this two-language problem. It enables developers and end-users from numerous fields to harness this combination of high productivity and high performance to help move ideas from conception to production at speeds that were not previously possible.

AUTHOR'S BIO

Avik Sengupta has worked on risk and trading systems in investment banking for many years, mostly using Java interspersed with snippets of the exotic R and K languages. This experience left him wondering whether there were better things out there. Avik's quest concluded with the appearance of Julia in 2012. He has been happily coding in Julia and contributing to it ever since.

AUTHOR'S BIO

Dr **Simon Byrne** is a quantitative software developer at Julia Computing, where he implements numerical routines for financial and statistical models. Simon has a PhD in Mathematics from the University of Cambridge and has extensive experience in computational statistics and machine learning in both academia and industry. He has been contributing to the Julia project since 2012.

Julia has been used in many sectors and industries, but financial services is one of its most natural habitats. It has been used by organisations large and small to calculate regulatory capital, value portfolios and design trading strategies. In this article, we will demonstrate one such example use case, calculating the arbitrage opportunities in FX cross rates. We will show how a complex optimisation problem can be implemented in a few lines of Julia code.

To start with, we will provide a brief introduction to Julia and its capabilities, as many readers might not yet be familiar with the language.

Julia's history

Julia began its life in 2009 as an academic collaboration among the project's creators Jeff Bezanson, Alan Edelman, Stefan Karpinski and Viral Shah to explore a solution to the two-language problem in technical computing. In February 2012, the project was announced publicly as an open-source project, freely downloadable and usable under an MIT license. Since that first public announcement, the Julia language community itself has grown to include over 500 contributors to the base language. It has spawned an ecosystem of over 1,300 packages in diverse topics including foundational mathematics (statistics, optimisation, differential equations), financial analytics, machine learning, bioinformatics, astrophysics, parallel and distributed computing, systems infrastructure (cloud management, identity management) and many other areas. The Julia user community currently numbers in the hundreds of thousands and is growing rapidly.

With the growing ecosystem and community, interest in the language has expanded beyond pure academic research. In 2015, Julia Computing, Inc. was founded by the creators based on industry requests for products, support and professional services centred on the Julia language ecosystem.

Julia's capabilities

Addressing the two-language problem necessitates careful design choices that balance the needs of the human and the computer. Julia's ability to achieve both high performance and high productivity involve the careful combination of the following features:

1. An expressive type system, allowing optional type annotations
2. Multiple dispatch using these types to select implementations
3. Metaprogramming for code generation
4. A dataflow type inference algorithm allowing types of most expressions to be inferred
5. Aggressive code specialisation against run-time types
6. Just-in-time (JIT) compilation using the low-level virtual machine (LLVM) compiler framework
7. Carefully written libraries that leverage the language design (points 1-6 above)

Points 1, 2 and 3 above are language features that are primarily concerned with how the human expresses their intentions in code, while points 4, 5 and 6 are mostly about language implementation and compiler internals addressing the needs of the computer. Point 7 involves bringing everything together to enable the development of high performance libraries in Julia. An extensive discussion of these points can be found in Bezanson et al. (2015) as well as references cited therein.

Performance

To demonstrate the type of performance that can be achieved with pure Julia code, we will perform a simple benchmark of the sum function, comparing Julia's performance to implementations in both C and Python. This particular example is derived from a lecture given by Prof Steven Johnson and refined by Prof David P. Sanders and Prof Alan Edelman. In this example, we show a simple computation that sums one million numbers. While we use and implement a function that is available in all standard libraries, this code is instructive in demonstrating the power of Julia's language design when using modern hardware.

To begin, let's create a sample vector of one million uniformly distributed random numbers on the interval from 0 to 1, and execute the version of the sum function included as part of a standard distribution of

Julia (see Listing 01).

click here to return to the top of the page