Zachary Krausman

COMP.3010.202

March 7, 2017

Julia

Introduction

Julia is “a high-level, high-performance dynamic programming language for technical computing” [BKS+ 12]. Stefan Karpinski was building software for his job and he was extremely annoyed by the fact that a specific project he was working on required the use of four different programming languages. The problem is that making software requires more than just making it work. A lot of things matter such as how fast a program runs and how effectively it uses memory. With that being said, it’s understandable that four languages may be required to maximize the efficiency and effectiveness of some software. The issue Stefan was having is that his codebase was a complete mess. Because of this and the fact that he was spreading bits and pieces of his software across multiple language, he found it very hard to debug and update his software. His goal was to create a language that could do everything relatively well. Additionally, Stefan noticed that languages like Ruby and Python made coding faster and easier but didn’t run as fast as language like C and Java [Fin14]. So, Stefan made some connections through friends and formed a team at his alma mater MIT to create the language. Many members of the team did a lot of math and wanted to make sure that Julia had advanced mathematical capabilities, they were tired of the tediousness of languages that already existed. The result is Julia, a language that is good at everything. Julia isn’t the best language at anything it doesn’t do anything better than anyone else. But it does virtually everything and at a reputable level. Previously when programmers wanted to combine two languages, in order to get the best of both worlds programmers had to use tools (usually third party) that would somehow combine or in some way link the code. With that being said, Julia doesn’t offer anything completely unique. There isn’t any one thing that Julia can do that can’t be done anywhere else, but in Julia programmers can do anything without having to glue languages together. This is achieved both by internal functionality and inherent connectedness to other languages meaning that to connect a lot of languages to Julia programmers don’t need to jump through hoops. The developers did some interesting and unique things to make this possible.

Type systems

The first thing this paper addresses is Julia’s type system. The definition of what a type system is seems to be much debated, but the purpose is simple. A type system in place to catch simple cases of when a programmer is trying to ask a program to do something it can’t do. When programmers ask programs to do things that the program can’t do, the program crashes. A type system is a first line of defense and is there to try to improve the chances that the program will finish of its own accord. Type systems are generally one of two types, either static or dynamic [Dyn15]. The main difference between the two is when they do their type checking. Type checking is making sure that the type of something being given to the program in some way is the same type that program is expecting to get. Dynamic typing means that the type check is done at runtime. What this means is that programmers can compile and therefore run a program which at some point will do something it clearly can’t do. Static typing means that the type check is done at compile time. In a statically typed language a compile won’t complete if the program doesn’t pass the type check. What this means is that programs that don’t pass the type check won’t get converted into an executable format. Julia has a dynamic type system that has some of the benefits of a static type system [BKS+e]. The main benefit of a dynamic typing system is that code can be pumped out at a much faster pace. The main benefit of dynamic typing is that writing programs is much faster because programmers don’t need to type define. What this means is that there’s less code and the code is less dense. This is good for fast implementation and rapid prototyping. The downside from a design standpoint is that programmers don’t have to think about the types and how they’re used which is often useful in improving efficiency. The main benefit of static typing is that since the type needs to be defined the program will be faster because it doesn’t have to figure out while it’s running what the types of every object is. This is good from a design standpoint because it allows for a much deeper understanding of how objects will be moving through a program. Forcing this understanding during initial design often makes the first iterations of programs written in a statically typed language more efficient than those in a dynamically typed language. Julia has the best of both worlds, it has type assertion. Programs in Julia don’t have to have explicit types because as previously stated it has a dynamic type system; however, if needed types can be asserted. Julia allows programmers to assert in two ways[Kar15]. The first is asserting a computation. Asserting a computation means that a programmer is telling the compiler that the result of a computation should be of a certain type if the result is not of a certain type, an error is thrown. This is somewhat between static and dynamic typing it doesn’t require that any variables in the computation be of that type it just requires that the result of some operations be that type. It’s important to note that asserting that the result of some computation be a specific type does not ensure that the computation will succeed and it does not mean that converting to the asserted type is assured. This is good for programmers because it allows them the speed of dynamic typing with the safety of static typing. Julia also allows for assertion on variables, which allows Julia to emulate the efficiency of statically typed languages (it uses type assertion for its built-in C compatibility) [BKS+e]. It allows for the assertion that variables are of a specific type and it holds them to that, just like a statically typed language if a variable is asserted to be of a certain type it must remain that type permanently. This is good because explicit assertion of variables does cause a lack of performance in dynamically typed languages. What this mean is that code can be implemented fast due to the lack of static typing but then later once a programmer knows the code works they can go back and add these assertions which the Julia compiler can use to increase the speed. When types are constantly changing, the program slows down to figure out exactly what’s changing, which as stated before is the major downside to dynamic typing but Julia allows programmers to get most of the performance back with its type assertions. Julia was created with the idea that a programming language should be able to do everything. This ability to type assert or not type assert allows the programmer to pick and choose how they want to develop their program. It gives options which is what Julia is all about.

Expressiveness (How clearly the code expresses the developer’s intentions)

The next thing that this paper addresses is Julia’s expressiveness. Expressiveness has a lot of context and even scopes it can be used in, so this paper will make sure to address the context and current definition that is being used [Cop10]. When talking about expressiveness in terms of writing it means how easily readable the code is both for humans and compilers[Cop10]. Historically, the biggest programming languages have been written by English speaking programmers. For non-English speakers, it means they are forced to use function names and variable names that might not make sense to them. From an expressive standpoint, this is bad because it restricts the amount of understanding that can come from simply reading code. Translation is hard which is why it would be very to have a languages functions be made available in different languages because then the compiler would have to understand that and there would likely have to be some way to translate code from language to language. Julia does take a good step though by adding Unicode support for its variable names which means that many of the Asian and African languages who use different alphabets can name variables in their native tongue [BKS+e]. Documentation is an aspect of expressiveness that often goes overlooked because it isn’t built-in to both languages. Despite this, it’s often integral in truly understanding the code and what exactly the code it doing. Julia has a documentation system which converts normal comments or tagged comments into Markdown formatted documentation (Markdown by default, other string macros can be implemented). The most important aspect of this is the fact that Julia has it built in and its system doesn’t require specific syntax or comment style to create documentation (like many third-party systems in other languages require). Expressiveness in terms of writing code refers to the variety of ideas that can be represented in a programming language and one of the ways this is represented in Julia is through its syntax. Syntax, at least stylistically is only guaranteed to be the same for things that are built into the language and could vary for resources created by a third party. Julia is no different, they don’t impose style guidelines on all their packages. However, Julia has a lot of functionality built into the language specifically in its mathematical capabilities. The developers didn’t write all the code for Julia’s math functionality for much of it they simply adapted it so it would work in Julia natively. For example, Julia’s linear algebra functions are implemented mostly by calling functions from LAPACK a FORTRAN math library (some are taken from other libraries) [BKS+b]. Julia has built in support for calling FORTRAN code, so why did the developers choose to integrate these functions into Julia instead of just allowing the user to call LAPACK themselves? There are two reasons, the main reason is that they wanted these functions to be natively available in Julia. This makes a lot of sense; FORTRAN was first released in the late 1950’s of course it’s been updated since then but in some senses, it’s still archaic. Just because it’s old doesn’t make it a bad language, the reason it’s been around so long is because there are things that it does well, like math. Julia was first released in 2012, it is a very current language and it has a lot of very current functionality. This is just one of those “don’t fix what isn’t broken” scenarios, the developers wanted to use the optimal implementations from LAPACK but also wanted to modernize their functionality hence adding them to Julia. The other reason is that the developers wanted to be able to pull from multiple sources, they wanted to be able to assemble their math libraries by what they judged to best on a function by function basis.

The relationship between speed and expressiveness

Speed and expressiveness don’t usually go hand in hand, for example look at the following code snippet of a basic loop shown in both Groovy and Java[Sli09].

Groovy Java

3.times { for(int i=0; i<3; i++) {

println 'Hip hip hooray'} System.out.println("Hip hip hooray");}

This is an extremely simple and short example so there may not be a discernable difference in running this code, but it illustrates the concept none the less. Java is a faster language than Groovy, but Groovy is much more expressive. It’s not to say that the Java code isn’t simple, it uses common syntax. The issue with the Java code is that it’s much more verbose. In the case of Java and Groovy, the tradeoffs are very clear when picking Groovy performance is given up and when picking Java readability is given up. Going back to Julia and FORTRAN, the question is whether this is still the case. Does Julia give up functionality and/or speed by being more expressive? The answer in most cases is no, and here is an example to prove it. To understand where Julia gets its performance let’s look at one of the benchmark tests written by Julia developers. This is the rand\_mat\_mul test which multiplies matrices filled with random numbers. The code for the Julia and Fortran implementations are shown below[Sha13a][Sha13b]. (Note about testing and compiling conditions: the FORTRAN version was written in FORTRAN 90 and compiled with gcc version 5.3 using -O3, both tests were compiled and ran on a 64-bit version of Windows, the C test was not run for comparison because it was designed for a Linux system. Both tests were run multiple times to ensure accuracy. It should also be noted that there is a lot more code required to run the FORTREN test, that code is not included below and is also not included when talking about performance but is called with the code call rand\_mat\_mul(1000, C) with 1000 being the dimensions that the empty matrix C will be.)

Julia FORTRAN

1. rand(1000,1000) \* rand(1000,1000) subroutine rand\_mat\_mul(n, C)

2. integer, intent(in) :: n

3. real(dp), intent(out), allocatable :: C(:, :)

4. real(dp), allocatable :: A(:, :), B(:, :)

5. allocate(A(n, n), B(n, n), C(n, n))

6. call random\_number(A)

7. call random\_number(B)

8. C = matmul(A, B)

9. end subroutine

What this code does is create two 1000 by 1000 matrices filled with random floating point numbers on the interval of [0,1) and it multiplies them. The Julia code is short because of dynamic typing (discussed above) which also plays into its implementation of rand() and how matrices(arrays) are created on top of all that Julia has defined behavior for its \* operator to multiply matrices (as long as the multiplication is possible and type conversion is available if necessary). The FORTRAN code makes sense and is long for the inverse reasons. Lines 2-4 are declarations which are required because FORTRAN is statically typed, line 5 is for allocation again because FORTRAN is statically typed it requires allocation (Julia does all its allocation automatically during compilation although it can do it manually through the use of C’s malloc() function.) Lines 6 and 7 are calls to fill the two matrices with random floating point number and line 8 is call to a function that multiplies the matrices together and stores the result in a matrix called C. As explained both programs do the same thing. The developers test shows the FORTRAN program running at one-third the speed of the Julia program [BKS+c]. The tests done for this paper find that the Julia code completes in ~220-250ms while the FORTRAN program completes in ~500-515ms. While this result is much better than the official test, the FORTRAN code still runs relatively slow. When looking at the disassembled code, it’s not hard to see why. The FORTRAN code breaks down into 260 lines of 32-bit assembly while Julia breaks down in 94 lines of 64-bit assembly. It should be noted that Intel makes a FORTRAN compiler that is around forty percent faster when compiled for efficiency but disregarding standards and is also not free for commercial use which is why it was not used for testing [Int16] (gfortran also has an option to compile for efficiency without caring about standards, that option was not used for this papers or Julia’s official tests). This is what it means for a language to be expressive from a purely technical standpoint. Programming is just writing code that gets broken down into simpler and simpler code until it reaches a hardware level of implementation. For compiled languages, the process is usually to translate high level code into simpler code (usually simpler versions of itself) and then eventually to translate very simplified code into assembly code. The assembly code is then converted to machine instructions by an assembler. In some cases, machine instructions are broken down into an even simpler language called microcode. Microcode is not in every architecture, is very varying in functionality and has circuit level compatibility. Because of this, machine code is going to be considered the lowest level for the purposes of this paper. Every function

call in high-level languages breaks down into varying but similar sets of assembly instructions based on type. Expressiveness isn’t exactly about how much assembly is generated per line but rather how much of the assembly is visible in the high-level code. The point is that languages don’t necessarily do everything expressively. In the example shown above, the Julia code is un-expressive. It generates 94 lines of assembly code, which contains around ten different sections which makes sense because that’s about how many things are happening behind the scenes. Within the rand call, there’s making the arrays, allocating space for them, generating numbers and then putting the numbers in. That happens twice, then there’s multiply operation which probably has checks involved. (Note, the previous statement saying ten sections in the assembly means ten operations is just an estimate as well as a vast oversimplification. It will not be like that all the time and future or past version of Julia may section the code differently. FORTRAN put the whole function in one 260-line block of code with no separation. That clearly doesn’t mean there was only one function call at high level it’s just how the compiler chose to format). It can be thought of like this, the most expressive code is code that can be explained in as many sentences as the code is long. Which just shows that expressive code is not always fast or efficient code. As stated before expressiveness is on a function to function basis, but when talking about the expressiveness of a language it’s more of the average. A ratio is a good way to think about this, how many “units” of functionality does one get per “unit” of code. Expressive code has a ratio that is close to one. Julia is always a powerful language, it can do a lot of things, and it can do them fast. However, due to it being a dynamically typed language a lot of functionality can be fit into a very small amount of code. What this means is that sometimes Julia can be un-expressive. There is always the option to write the code the long way and to type-assert which will give the code a statically typed level of expressiveness. Choosing how to write the code should stem from who will be reading it and how fast the program needs be written as well as how fast it must run. If development speed matters, write the shortest code (usually will have a lot of functionality per line). If run speed matters, write the most efficient code (usually longer and has a good ratio of functionality to line). If other people are going to be reading it programmers should cater to the most prevalent knowledge level e.g. if the program is going to be read by the beginners make sure it’s simple (programs geared towards mid-tier knowledge are the most expressive). In closing, I think the most important thing is that Julia gives more options in terms of how expressive one can be. Other languages don’t have a wide range of how expressive the code can be, especially statically typed ones. Julia does, and it stems from the fact that it can essentially be dynamically or statically typed. As stated before Julia gives programmers options but its variety of expressiveness is one of the ways in which Julia programmers give up less when making decisions or at the very least are much more aware of what they may be giving up.

Coroutines

Before going into what exactly a coroutine is, it’s more important to define a problem they solve. Think about a bakery it makes food and people buy it, keep in mind that there is a limited amount of shelf space. So, the problem is that bakers don’t stop baking until they’re told. The bakers won’t know to stop baking until someone tells them the shelves are full. The customers can’t buy anything if there’s no product. This all seems very logical from a human standpoint, but that isn’t how programs work. Programs do what you tell them to do, not what you want them to do. Programmers need to tell their programs that the “shelves are full” and that the “shelves are empty”. This is what’s known as the producer consumer problem. There is a limited number of space and things are being taken in and out of that space, as a programmer one needs to be sure that if something is trying put a value in the space, the space isn’t full and if something is trying to take a value out of the space there needs to be something to take out. The action of putting something in is known as producing and the action of taking something out is known as consuming. The problem is that programs shouldn’t be able to produce into somewhere with no space and consumers shouldn’t be able to consume when nothing is there. The solution in a general sense is to keep track of how many things are being produced and how many things are being consumed and to block consumers who are trying to produce more than the initial size of the space plus the number of things that have been consumed. Then for the consumers it’s to make sure that they don’t consume more than the number of things produced. Julia has built-in functionality that provides somewhat of a solution to this problem, they have a produce() and consume() function [BKS+a]. The produce() function takes some value then suspends that routine and hands control over the a consumer routine. The consume() function take the value returns it and then returns control to the producer routine. This is not ideal functionality because it doesn’t implicitly handle the situation in which multiple things can be produced before something needs to be consumed and vice versa. The Julia developers made this for mathematical use when some computations will need to pass values back and forth. It doesn’t implicitly handle the producer-consumer problem when space is limited it only handles the control switch and passing a value back and forth nor does it implicitly allow multi-process synchronization. The important thing is that Julia has all it needs to for the solution to be assembled. Using some integers as counters (no limit on values unless multiple producers or consumers in which case binary ones will be needed) and some of the underlying functions that produce() and consume() call, a good solution is available. Julia has the yield(), yieldto(), wait(), notify(), Condition() [BKS+e]. It would take too long to explain what all of them do, but the names are self-explanatory. Essentially these functions give us the ability

to switch conditionally and to create a block queue containing producers waiting for their turn, but as of right now that would have to be implemented by the programmer. C has a good way of handling this using various data structures like event counters, sequencers and semaphores. With the functionality, already available in Julia these efficient data structures and functions for them could easily be created and in a future update hopefully they will be implemented. The thing that Julia has going for it is portability which comes from the fact that the functionality is built-in. The C library for implementing the data structures I just mentioned is only available on UNIX systems, Windows handles threads differently and therefore has its own library for handling this kind of stuff. This means that C code using either of these libraries is not cross compatible. Now of course if the functionality were the in Julia, the Linux and Windows implementations would be different under the hood but the programs would be compatible on both operating systems. Julia has all the necessary tools (including parallel computing which was not talked about but is often associated with the producer-consumer problem) to have a good coroutine system but it’s current state is only for mathematical usage. It should be noted that Julia also has the tools the solve the reader-writer problem which is somewhat related to the producer-consumer problem [BKS+e]. This is because the solution to the reader-writer problem uses the same tools as the ones for the producer-consumer problem.

Conclusion

Julia is a language developed by scientists and mathematicians for scientists and mathematicians [BKS+ 12]. I’m of the opinion that right now there are some things it doesn’t do well, even within the realm of math. To its credit however, the language has been out for less than five years. C is one of the fastest languages, it’s one of the fastest because it’s been out for so long and it’s just been optimized and optimized over and over. Julia’s benchmarks also show FORTRAN being a very fast language [BKS+c]. C and FORTRAN have been around for ten times longer than Julia. If Julia is able to maintain development for the next fifty years I wholeheartedly believe that it will be the next C. But that’s it’s biggest problem is that just like everything else it’s a business. Programming languages are a business and I’m sure good programming languages died too young because they didn’t find a way to keep developing, or in more simple terms they didn’t find a way to make money. The Julia development team has started Julia Computing LLC to “provide training, commercial support and consulting for those who want to use the language” [Nov15]. Right now, the company offers four products that essentially just allow people (usually companies) to use Julia on larger scales. This is the right way to commercialize a programming language. Developers aren’t going to get users, if users are required to pay for a programming language the idea its self is ridiculous. But the ability to stretch its use cases and most importantly can provide support for money makes a lot of sense for the developers and for companies. If I was using a programming language in my company I would certainly like the developers to be my help line. Another thing that makes Julia unique is that they’ve accepted the fact that there are things they will never do better than other languages and if you can’t beat them, join them. As stated earlier, C and FORTRAN are two of the fastest languages and Julia has built-in support for using code from both languages and for C it even has the ability to intermingle C code with Julia code. They know that there are some things about C and FORTRAN that just are the best and there’s no reason to implement Julia versions that won’t be better. I think that as of right now Julia is not the language for me or really for any computer science student because it’s not a language that will get a student or recent grad a job right now. With that being said, I believe that Julia will continue grow and hopefully be adopted more and more because if it keeps growing at the pace it has, it will easily become the language of my generation.

|  |  |
| --- | --- |
| References  [BKS+a] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, and et al. Control flow.  [BKS+b] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, and et al. Linear algebra.  [BKS+c] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, and et al. A summary of features.  [BKS+d] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, and et al. Tasks and parallel computing.  [BKS+e] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, and et al. Types.  [BKS+12] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, and | |
|  | et al. Why we created julia, Feb 2012. |
| [Cle17] | Julia — definition, programming, history, 2017. |
| [Cop10] | Reed Copsey. What do you mean by the expressiveness of a programming language?, Mar 2010. |
| [Dyn15] | Dynamic typing vs. static typing, Mar 2015. |
| [Fer12] | Steve Ferg. Static vs. dynamic typing of programming languages, Apr 2012. |
| [Fin14] | Klint Finley. Out in the open: Man creates one programming language to rule them all, Feb 2014. |
| [Hal14] | Matt Hall. Julia in a nutshell, Sep 2014. |
| [Int16] | Intel fortran compilers details, Sep 2016. |
| [Kar15] | Stefan Karpinski. Is julia dynamically typed?, Jan 2015. |
| [Nov15] | Jordan Novet. Why the creators of the julia programming language just launched a startup, May 2015. |
| [Sha13a] | Viral B Shah. Julialang/fortran, Jul 2013. |
| [Sha13b] | Viral B Shah. Julialang/julia, Jul 2013. |
| [sli09] | slim. What does ”expressive” mean when referring to programming languages?, Mar 2009. |