WhyJulia

September 11, 2018

1 Why Does Julia Work So Well?

There is an obvious reason to choose Julia:

> it's faster than other scripting languages, allowing you to have the rapid development of Python/MATLAB/R while producing code that is as fast as C/Fortran

Newcomers to Julia might be a little wary of that statement.

1. Why not just make other scripting languages faster? If Julia can do it, why can't others?
2. How do you interpert Julia benchmarks to confirm this? (This is surprisingly difficult for many!)
3. That sounds like it violates the No-Free-Lunch heuristic. Is there really nothing lost?

Many people believe Julia is fast **because it is Just-In-Time (JIT) compiled** (i.e. every statement is run using compiled functions which are either compiled right before they are used, or cached compilations from before). This leads to questions about what Julia gives over JIT'd implementations of Python/R (and MATLAB by default uses a JIT). These JIT compilers have been optimized for far longer than Julia, so why should we be crazy and believe that somehow Julia quickly out-optimized all of them? However, that is a complete misunderstanding of Julia. What I want show, in a very visual way, is that Julia is fast because of its design decisions. The core design decision, **type-stability through specialization via multiple-dispatch** is what allows Julia to be very easy for a compiler to make into efficient code, but also allow the code to be very concise and "look like a scripting language". This will lead to some very clear performance gains.

But what we will see in this example is that Julia does not always act like other scripting languages. There are some "lunches lost" that we will have to understand. Understanding how this design decision effects the way you must code is crucial to producing efficient Julia code.

To see the difference, we only need to go as far as basic math.

1.1 Arithmetic in Julia

In general, math in Julia looks the same as in other scripting languages. One detail to note is that the numbers are "true numbers", as in a `Float64` is truly the same thing as a 64-bit floating point number or a "double" in C. A `Vector{Float64}` is the same memory layout as an array of doubles in C, both making interop with C easy (indeed, in some sense "Julia is a layer on top of C") and it leads to high performance (the same is true for NumPy arrays).

Some math in Julia:

```
In [1]: a = 2+2
        b = a/3
        c = a⌀3 #\div tab completion, means integer division
        d = 4*5
        println([a;b;c;d])
```

```
[4.0, 1.33333, 1.0, 20.0]
```

Note here that I showed off Julia's unicode tab completion. Julia allows for unicode characters, and these can be used by tab completing Latex-like statements. Also, multiplication by a number is allowed without the * if followed by a variable. For example, the following is allowed Julia code:

```
In [2]:  = 0.5
        f(u) = *u; f(2)
        sin(2)
```

```
Out[2]: -2.4492935982947064e-16
```

1.2 Type-stability and Code Introspection

Type stability is the idea that there is only 1 possible type which can be outputttted from a method. For example, the reasonable type to output from `*(::Float64,::Float64)` is a `Float64`. No matter what you give it, it will spit out a `Float64`. This right here is multiple-dispatch: the `*` operator calls a different method depending on the types that it sees. When it sees floats, it will spit out floats. Julia provides code introspection macros so that way you can see what your code actually compiles to. Thus Julia is not just a scripting language, it's a scripting language which lets you deal with assembly! Julia, like many languages, compiles to LLVM (LLVM is a type of portable assembly language).

```
In [3]: @code_llvm 2*5
```

```
; Function *
; Location: int.jl:54
define i64 @"julia_*_33751"(i64, i64) {
top:
  %2 = mul i64 %1, %0
  ret i64 %2
}
```

This output is saying that a floating point multiplication operation is performed and the answer is returned. We can even look at the assembly

```
In [4]: @code_native 2*5
```

```
        .text
; Function * {
; Location: int.jl:54
        imulq        %rsi, %rdi
        movq        %rdi, %rax
        retq
        nopl        (%rax,%rax)
;}
```

This shows us that the * function has compiled down to exactly the same operation as what happens in C/Fortran, meaning it achieves the same performance (even though it's defined in Julia). Thus it is possible to not just get "close" to C, but actually get the same C code out. In what cases does this happen?

The interesting thing about Julia is that, asking which cases this happens is not the right question. the right question is, in what cases does the code not compile to something as efficient as C/Fortran? The key here is type-stability. If a function is type-stable, then the compiler can know what the type will be at all points in the function and smartly optimize it to the same assembly as C/Fortran. If it is not type-stable, Julia has to add expensive "boxing" to ensure types are found/known before operations.

This is the key difference between Julia and other scripting languages The upside is that Julia's functions, when type stable, are essentially C/Fortran functions. Thus ^ (exponentiation) is fast. However, `^(::Int64,::Int64)` is type-stable, so what type should it output?

```
In [5]: 2^5
```

```
Out[5]: 32
```

```
In [6]: 2^-5
```

```
Out[6]: 0.03125
```

Here we get an error. In order to guarantee to the compiler that ^ will give an Int64 back, it has to throw an error. If you do this in MATLAB, Python, or R, it will not throw an error. That is because those languages do not have their entire language built around type stability.

What happens when we don't have type stability? Let's inspect this code:

```
In [7]: @code_native ^(2,5)
```

```
        .text
; Function ^ {
; Location: intfuncs.jl:220
        pushq        %rax
        movabsq        $power_by_squaring, %rax
        callq        *%rax
        popq        %rcx
        retq
        nop
;}
```

Now let's define our own exponentiation on integers. Let's make it "safe" like the form seen in other scripting languages:

```
In [8]: function expo(x,y)
            if y>0
                return x^y
            else
                x = convert(Float64,x)
                return x^y
            end
        end
```

```
Out[8]: expo (generic function with 1 method)
```

Let's make sure it works:

```
In [9]: println(expo(2,5))
        expo(2,-5)
```

```
32
```

```
Out[9]: 0.03125
```

What happens if we inspect this code?

```
In [10]: @code_native expo(2,5)
```

```
        .text
; Function expo {
; Location: In[8]:2
        pushq   %rbx
        movq    %rdi, %rbx
; Function >; {
; Location: operators.jl:286
; Function <; {
; Location: int.jl:49
        testq   %rdx, %rdx
;}}
        jle     L36
; Location: In[8]:3
; Function ^; {
; Location: intfuncs.jl:220
        movabsq $power_by_squaring, %rax
        movq    %rsi, %rdi
        movq    %rdx, %rsi
        callq   *%rax
;}
        movq    %rax, (%rbx)
```

```
        movb        $2, %dl
        xorl        %eax, %eax
        popq        %rbx
        retq
; Location: In[8]:5
; Function convert; {
; Location: number.jl:7
; Function Type; {
; Location: float.jl:60
L36:
        vcvtsi2sdq        %rsi, %xmm0, %xmm0
;}}
; Location: In[8]:6
; Function ^; {
; Location: math.jl:780
; Function Type; {
; Location: float.jl:60
        vcvtsi2sdq        %rdx, %xmm1, %xmm1
        movabsq        $__pow, %rax
;}
        callq        *%rax
;}
        vmovsd        %xmm0, (%rbx)
        movb        $1, %dl
        xorl        %eax, %eax
; Location: In[8]:3
        popq        %rbx
        retq
        nopw        %cs:(%rax,%rax)
;}
```

That's a very visual demonstration on why Julia achieves such higher performance than other scripting languages by how it uses type inference.

2 Core Idea: Multiple Dispatch + Type Stability => Speed + Readability

Type stability is one crucial feature which separates Julia apart from other scripting languages. In fact, the core idea of Julia is the following statement:

Multiple dispatch allows for a language to dispatch function calls onto type-stable functions. This is what Julia is all about, so let's take some time to dig into it.If you have type stability inside of a function (meaning, any function call within the function is also type-stable), then the compiler can know the types of the variables at every step. Therefore it can compile the function with the full amount of optimizations since at this point the code is essentially the same as C/Fortran code. Multiple-dispatch works into this story because it means that $*$ can be a type-stable function: it

just means different things for different inputs. But if the compiler can know the types of `a` and `b` before calling `*`, then it knows which `*` method to use, and therefore it knows the output type of `c=a*b`. Thus it can propogate the type information all the way down, knowing all of the types along the way, allowing for full optimiziations. Multiple dispatch allows `*` to mean the "right thing" every time you use it, almost magically allowing this optimization.

There are a few things we learn from this. For one, in order to achieve this level of optimization, you must have type-stability. This is not featured in the standard libraries of most languages, and was choice that was made to make the experience a little easier for users. Secondly, multiple dispatch was required to be able to specialize the functions for types which allows for the scripting language syntax to be "more explicit than meets the eye". Lastly, a robust type system is required. In order to build the type-unstable exponentiation (which may be needed) we needed functionalities like convert. Thus the language must be designed to be type-stable with multiple dispatch and centered around a robust type system in order to achieve this raw performance while maintaining the syntax/ease-of-use of a scripting language. You can put a JIT on Python, but to really make it Julia, you would have to design it to be Julia.

2.1 The Julia Benchmarks

The Julia benchmarks, featured on the Julia website, test components of the programming language for speed. **This doesn't mean it's testing the fastest implemention**. That is where a major misconception occurs. You'll have an R programmer look at the R code for the Fibonacci calculator and say "wow, that's terrible R code. You're not supposed to use recursion in R. Of course it's slow". However, the Fibonacci problem is designed to test recursion, not the fastest implementation to the the ith Fibonacci number. The other problems are the same way: testing basic components of the langauge to see how fast they are.

Julia is built up using multiple-dispatch on type-stable functions. As a result, even the earliest versions of Julia were easy for compilers to optimize to C/Fortran efficiency. It's clear that in almost every case Julia is close to C. Where it is not close to C actually has a few details. The first is the Fibonacci problem where Julia is 2.11x from C. This is because it is a test of recursion, and Julia does not fully optimize recursion (but still does very well on this problem!). The optimization which is used to receive the fastest times for this type of problem is known as Tail-Call Optimization. Julia can at any time add this optimization, though there are reasons why they choose not to. The main reason is: any case where tail-call optimization is possible, a loop can also be used. But a loop is also more robust for optimizations (there are many recursive calls which will fail to tail-call optimize) and thus they want to just recommend using loops instead of using fragile TCO.

The other cases where Julia doesn't do as well are the `rand_mat_stat` and the `parse_int` tests. However, this is largely due to a feature known as bounds checking. In most scripting languages, you will receive an error if you try to index an array outside of its bounds. Julia will do this by default:

```
In [11]: function test1()
             a = zeros(3)
             for i=1:4
                 a[i] = i
             end
         end
         test1()
```

```
BoundsError: attempt to access 3-element Array{Float64,1} at index [4]



Stacktrace:

 [1] setindex! at ./array.jl:769 [inlined]

 [2] test1() at ./In[11]:4

 [3] top-level scope at In[11]:7
```

However, Julia allows you to turn this off using the `@inbounds` macro:

```
In [12]: function test2()
             a = zeros(3)
             @inbounds for i=1:4
                 a[i] = i
             end
         end
         test2()
```

This gives you the same unsafe behavior as C/Fortran, but also the same speed (indeed, if you add these to the benchmarks they will speed up close to C). This is another interesting feature of Julia: it lets you **by default have the safety of a scripting language, but turn off these features when necessary (/after testing and debugging) to get full performance.**

3 Small Expansion of the Idea: Strict Typing

Type-stability is not the only necessity. You also need strict typing. In Python you can put anything into an array. In Julia, you can only put types of `T` into a `Vector{T}`. To give generality, Julia offers various non-strict forms of types. The biggest example is `Any`. Anything satisfies `T:<Any` (hence the name). Therefore, if you need it, you can create a `Vector{Any}`. For example:

```
In [14]: a = Vector{Any}(undef,3)
         a[1] = 1.0
         a[2] = "hi!"
         a[3] = :Symbolic
         a

Out[14]: 3-element Array{Any,1}:
          1.0
           "hi!"
           :Symbolic
```

A less extreme form of an abstract type is a Union type, which is just what it sounds like. For example:

```
In [15]: a = Vector{Union{Float64,Int}}(undef,3)
         a[1] = 1.0
         a[2] = 3
         a[3] = 1/4
         a

Out[15]: 3-element Array{Union{Float64, Int64},1}:
          1.0
          3
          0.25
```

This will only accept floating point numbers and integers. However, it is still an abstract type. A function which is called on an abstract type cannot know the type of any element (since, in this example, any element can be either a float or an integer). Thus the optimization that was achieved by multiple-dispatch, knowing the type each step of the way, is no longer present. Therefore the optimizations are gone and Julia will slow down to the speed of other scripting languages.

This leads to the performance principle: use strict typing whenever possible. There are other advantages: a strictly typed `Vector{Float64}` is actually byte-compatible with C/Fortran, and so it can be used directly by C/Fortran programs without conversion.

4 Lunch Money

It's clear that Julia made clever design decisions in order to achieve its performance goals while still being a scripting language. However, what exactly is lost? Next I will show you a few peculiarities of Julia that come from this design decision, and the tools Julia gives you to handle them.

4.1 Performance as Optional

One thing I already showed is that Julia gives many ways to achieve high performance (like `@inbounds`), but they don't have to be used. You can write type-unstable functions. It will be as slow as MATLAB/R/Python, but you can do it. In places where you don't need the best performance, it's nice to have this as an option.

4.2 Checking for Type-Stability

Since type-stability is so essential, Julia gives you tools to check that your functions are type stable. The most important is the `@code_warntype` macro. Let's use it to check a type-stable function:

```
In [16]: @code_warntype 2^5

Body::Int64
220 1  %1 = invoke Base.power_by_squaring(_2::Int64, _3::Int64)::Int64
          return %1
```

Notice that it shows all of the variables in the function as strictly typed. What about in our `expo`?

```
In [17]: @code_warntype expo(2,5)
```

```
Body::Union{Float64, Int64}
 >2 1  %1  = (Base.slt_int)(0, y)::Bool
           goto #3 if not %1
  3 2  %3  =  (x, Int64)
  ^     %4  = invoke Base.power_by_squaring(%3::Int64, _3::Int64)::Int64
           return %4
  5 3  %6  =  (x, Int64)
  Type     %7  = (Base.sitofp)(Float64, %6)::Float64
  6    %8  =  (%7, Float64)
  ^     %9  = (Base.sitofp)(Float64, y)::Float64
      %10 = $(Expr(:foreigncall, "llvm.pow.f64", Float64, svec(Float64, Float64), :(:llvmcall)
           return %10
```

Notice that possible returns are the temporary `%4` and `%10` which are different types, and so the return type is inferred as `Union{Float64,Int64}`. To trace exactly to where this instability occurs, we can use Traceur.jl:

```
In [19]: using Traceur
         @trace expo(2,5)

 Warning: x is assigned as Int64
 @ In[8]:2
 Warning: x is assigned as Float64
 @ In[8]:5
 Warning: expo returns Union{Float64, Int64}
 @ In[8]:2


Out[19]: 32
```

This tells us that on line 2 `x` is assigned to an `Int` while on line 5 it's assigned to a `Float64`, and so it's inferred as `Union{Float64, Int64}`. Line 5 is where we put the explicit `convert` call, so this identified exactly the issue for us.

4.3 Dealing With Necessary Type-Instabilities

For one, I already showed that some functions will error while in other scripting languages they will "read your mind". In many cases you will realize that you could've just used a different type from the start and achieved type-stability (why not just do `2.0^-5`?). However, there are some cases where you won't find an appropriate type. This can be easily fixed by conversions, though you then lose type stability. Instead you have to think about your design and cleverly use multiple dispatch.

So let's say that we have `a` as a `Vector{Union{Float64,Int}}`. We may run into a case where we have to use `a`. Assume that on each element of `a` we have to perform a lot of operations. In this case, knowing the type of a given element will lead to massive performance gains, but since it's in a `Vector{Union{Float64,Int}}`, they will not be known in a function like:

```
In [17]: function foo(array)
           for i in eachindex(array)
             val = array[i]
             # do algorithm X on val
           end
         end

Out[17]: foo (generic function with 1 method)
```

However, we can fix this with multiple dispatch. We can write a dispatch on elements:

```
In [18]: function inner_foo(val)
           # Do algorithm X on val
         end

Out[18]: inner_foo (generic function with 1 method)
```

and instead define foo as:

```
In [20]: function foo2(array::Array)
           for i in eachindex(array)
             inner_foo(array[i])
           end
         end

Out[20]: foo2 (generic function with 1 method)
```

Since types are checked for dispatch, the function `inner_foo` is strictly typed. Thus if `inner_foo` is type-stable, then we can achieve high performance by allowing it to specialize within `inner_foo`. This leads to a general design principle that, if you're dealing with odd/non-strict types, you can use an outer function to handle the type logic while using an inner function for all of the hard calculations and achieve close to optimal performance while still having the generic abilities of a scripting language.

4.4 REPL "Globals" Have Bad Performance

Globals in Julia have awful performance. Not using globals is the first fact in the Performance Tips. However, what newcommers don't realize is that the REPL is the global scope. To see why, recall that Julia has nested scopes. For example, if you have a function inside of a function, then the inner function has all of the variables of the outer function.

```
In [21]: function test(x)
             y = x+2
             function test2()
                 y+3
             end
             test2()
         end

Out[21]: test (generic function with 1 method)
```

In `test2`, `y` is known because it is defined in `test`. This will all work to give something performant if `y` is type-stable since `test2` could then assume that `y` is always an integer. But now look at what happens at the highest scope (and thus effectively the global scope):

```
In [22]: a = 3
         function badidea()
             a + 2
         end
         a = 3.0

Out[22]: 3.0
```

Because no dispatch is used to specialize `badidea`, and we can change the type of `a` at any time, and therefore `badidea` cannot add optimizations when compiling since the type of `a` is unknown during compile time. However, Julia allows us to specify variables as constant:

```
In [23]: const a_cons = 3
         function badidea()
             a_cons + 2
         end

Out[23]: badidea (generic function with 1 method)
```

Beware that functions will specialize using the value of the constants, so they should go unchanged after being set.

This will show up when trying to do benchmarks. The most common human error to see is for newcomers to benchmark Julia like:

```
In [25]: a = 3.0
         @time for i = 1:4
             global a
             a += i
         end

  0.000006 seconds (4 allocations: 64 bytes)
```

However, if we put this in a function, it will optimize (in fact, it will optimize away the loop and stick in the answer)

```
In [26]: function timetest()
             a = 3.0
             @time for i = 1:4
                 a += i
             end
         end
         timetest() # First time compiles
         timetest()
```

```
0.000001 seconds
0.000000 seconds
```

This is a very easy problem to fall for: don't benchmark or time things in the REPL's global scope. Always wrap things in a function or declare them as const. There is a developer thread to make the global performance less awful but, given the information from this notebook, you can already see that it will never be "not awful", it will just be "less awful".

5 Conclusion

Julia is fast by design. Type stability and multiple dispatch is necessary to do the specialization that is involved in Julia's compilation to make it work so well. The robust type system is required to make working with types at such a fine level in order to effectively achieve type-stability whenever possible, and manage optimizations when it's not totally possible.

5.1 Discussion / Project

Here's a good learning project: how would you design a new type `EasyFloats` to build MATLAB/Python/R arithmetic into Julia? How would you designed "arrays with NAs" to mimic R? Time the results and see what the difference from optimal is.