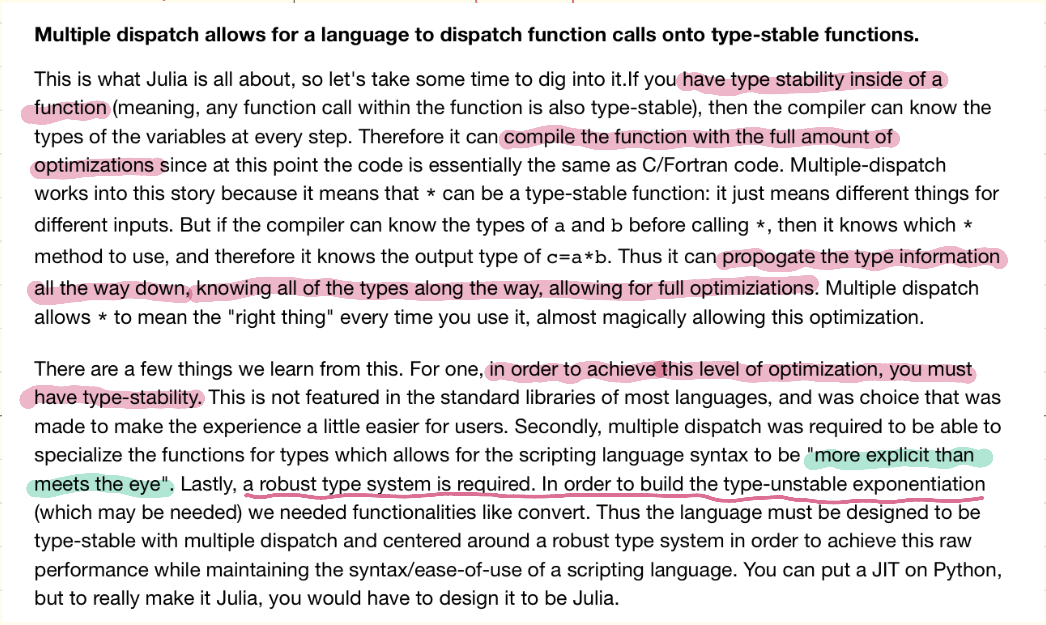
type-stability through multiple-dispatch

1. The numbers are true numbers, the same as in C
   ↳ interop with C is easy
2. type-stability → there is only 1 possible type that can be outputted from a method.

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.



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.

Julia is built-up using multiple-dispatch on type-stable functions.

Any case where tail-call optimization is possible. a loop can also be used. ← Tail-Call optimization for recursion

But a loop is also more robust for optimizations. recommend using loops instead of using fragile TCO.

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.**

no-free-lunch. What is lost in Julia?

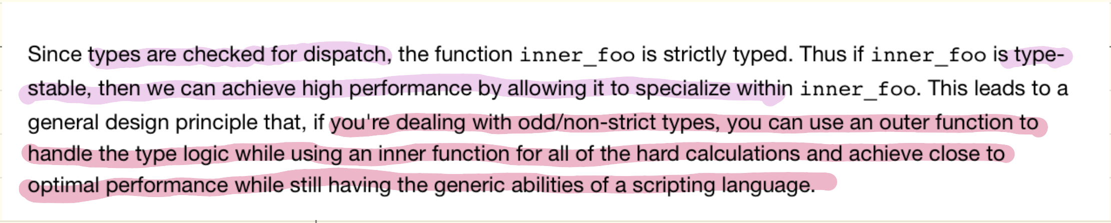
1. Performance as Optional

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.

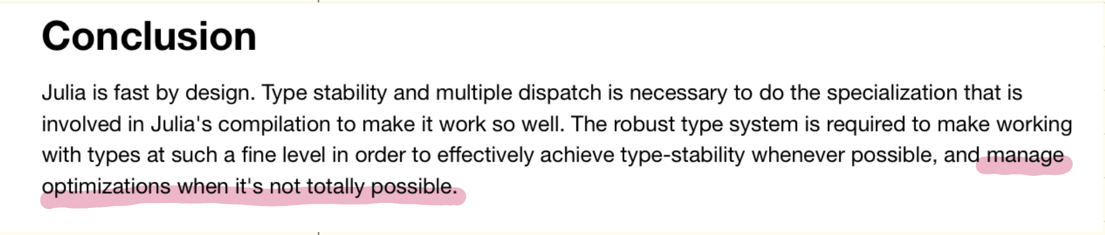
2. Checking for Type-Stability

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:

3. dealing with necessary type-instabilities


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.

3. Globals in Julia have awful performance.
Don't benchmark or time things in REPL's global scope.
Always wrap things in a function or declare them as const.


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.