10b. Macros as a Means for Optimizations

Martin Alfaro
PhD in Economics

INTRODUCTION

Customized computational approaches often have an edge over general-purpose built-in solutions, as they can tackle the unique challenges of a given scenario. However, the complexity of specialized techniques often deters their adoption among practitioners, who may lack the necessary expertise to implement them.

Macros offer a practical solution to bridge this gap, making specialized computational approaches more accessible to users. They're particularly well-suited for this purpose, due to their ability to take entire code blocks as inputs and transform them into an optimized execution approach. This capability allows practitioners to benefit from specialized algorithms, without having to implement them themselves.

In the upcoming sections, the role of macros in boosting performance will be central. By leveraging them, we'll be able to effectively separate the benefits provided by an algorithm from its actual implementation details. This decoupling will let us shift our focus from the nitty-gritty details of how to implement algorithms, to the more practical question of when to apply them. The current section in particular will concentrate on the procedure for applying macros, with a special emphasis on some subtle considerations arising in practice.

ABOUT MACROS FOR OPTIMIZATIONS

Macros bear a resemblance to functions in that they take an input and return an output. Their primary difference lies in that macros take an entire code block as their input, possibly yielding another code block as its output.

This unique feature enables macros to be applied for tasks that functions can't handle. One common application is **code simplification**. By automating repetitive tasks and eliminating redundant code, macros are capable of significantly improving code readability. For instance, suppose a function requires multiple slices of `x` to be converted into views. Without macros, this would involve repeatedly invoking `view(x, <indices>)`, resulting in verbose and error-prone code. Instead, prepending the function definition with `@views` will automatically handle all the slice conversions for us.

Another application of macros is **to modify how operations are computed**, which is the focus of this section. This functionality allows developers to package sophisticated optimization techniques, making advanced solutions accessible to users. Thus, users who might not be familiar with the underlying complexities of the method, only need to focus on selecting the most suitable computational approach, rather than grappling with implementation details.

While macros are powerful tools, they're not without their limitations. Their black-box nature means that misuse can lead to unexpected results or compromise computational safety. That's why identifying suitable scenarios for each macro is crucial. While this requires a bit of upfront work, it's considerably less demanding than implementing the same functionality from scratch.

APPLYING MACROS IN FOR-LOOPS: @INBOUNDS AS AN EXAMPLE

One distinctive feature of Julia is its ability to execute for-loops with exceptional speed. In fact, carefully optimized for-loops tend to reach the highest possible performance within the language. This efficiency stems from the versatility of for-loops, which lets users fine-tune them for their specific needs. As a result, it's no surprise that one prominent application of macros is to implement specific computational approaches for for-loops.

To illustrate, let's consider the `@inbounds` macros. Although strictly speaking this doesn't implement a new computational approach, it does modify how for-loops are executed. Additionally, it's simple enough to easily demonstrate this role of macros.

To appreciate the impact of `@inbounds`, we first need to understand how for-loops typically behave in Julia. By default, the language implements **bounds checking**: when an element `x[i]` is accessed during the *i*-th iteration, Julia verifies that *i* falls within the valid range of indices for `x`. This built-in mechanism safeguards against errors and security issues caused by out-of-bounds access.

While bounds checking prevents bugs, it comes at a performance cost: these additional checks not only introduce computational overhead, but also limit the compiler's ability to implement certain optimizations. However, there are situations where iterations are guaranteed to stay within array bounds. In those cases, we can safely boost performance by disabling bounds checking through the `@inbounds` macro.

> **Trade-Offs Entailed by @inbounds**
>
> The `@inbounds` macro perfectly illustrates both the power and risks associated with macro usage. When applied judiciously, it can yield substantial performance gains, especially when multiple slices are involved.
>
> However, disabling bounds checking simultaneously renders code unsafe: it increases the risk of crashes and silent errors, additionally creating security vulnerabilities. In this context, `@inbounds` shifts the responsibility of applying the macro onto the user, who must be absolutely certain that the iteration range is within the arrays' bounds.

ILLUSTRATING @INBOUNDS

Broadly speaking, using a macro within a for-loop to modify its computational approach requires its addition at the beginning of the for-loop. For instance, to disable bounds checking for every indexed element within a for-loop, we simply need to prepend the for-loop with `@inbounds`. We can alternatively apply `@inbounds` individually to any specific line within the loop. Nonetheless, this possibility is specific to `@inbounds`, only arising because the macro can actually be employed even outside for-loops.

```julia
x = rand(1_000)

function foo(x)
    output = 0.

    @inbounds for i in eachindex(x)
                  a       = log(x[i])
                  b       = exp(x[i])
                  output += a / b
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  5.002 μs (0 allocations: 0 bytes)
```

```julia
x = rand(1_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        @inbounds a       = log(x[i])
        @inbounds b       = exp(x[i])
                  output += a / b
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  5.093 μs (0 allocations: 0 bytes)
```

The performance advantages of `@inbounds` don't only come from the elimination of bounds checking itself. Bounds checking is a form of conditional, where the iteration is executed contingent on all indices being within range. In the next sections, we'll see that conditional statements commonly limit the compiler's ability to apply further optimizations. Once we remove these checks, you give the compiler more leeway to implement additional enhancements.

To illustrate such possibility, the next example shows that the application of `@inbounds` triggers the so-called SIMD instructions. They're a form of parallelism within a core and will be explored in the upcoming sections.

```julia
v,w,x,y = (rand(100_000) for _ in 1:4)        # it assigns random vectors to v,w,x,y

function foo(v, w, x, y)
    output = 0.0

    for i in 2:length(v)-1
        output += v[i-1] / v[i+1] / w[i-1] * w[i+1] + x[i-1] * x[i+1] / y[i-1] * y[i+1]
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  231.242 μs (0 allocations: 0 bytes)
```

```julia
v,w,x,y = (rand(100_000) for _ in 1:4)        # it assigns random vectors to v,w,x,y

function foo(v, w, x, y)
    output = 0.0

    @inbounds for i in 2:length(v)-1
        output += v[i-1] / v[i+1] / w[i-1] * w[i+1] + x[i-1] * x[i+1] / y[i-1] * y[i+1]
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  154.179 μs (0 allocations: 0 bytes)
```

Warning! - Function Calls in For-Loop Bodies Can Disable Macro Effects

The use of functions without direct reference to slices could prevent the application of `@inbounds`. This can be observed below, where we compare approaches with and without `@inbounds` when a function is involved.

```julia
v,w,x,y = (rand(100_000) for _ in 1:4) # it assigns random
vectors to v, w, x, y
compute(i, v,w,x,y) = v[i-1] / v[i+1] / w[i-1] * w[i+1] +
                      x[i-1] * x[i+1] / y[i-1] * y[i+1]

function foo(v,w,x,y)
    output = 0.0

    @inbounds for i in 2:length(v)-1
        output += compute(i, v,w,x,y)

    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  271.897 μs (0 allocations: 0 bytes)
```

```julia
v,w,x,y = (rand(100_000) for _ in 1:4) # it assigns random
vectors to v, w, x, y



function foo(v,w,x,y)
    output = 0.0

    @inbounds for i in 2:length(v)-1
        output += v[i-1] / v[i+1] / w[i-1] * w[i+1] +
                  x[i-1] * x[i+1] / y[i-1] * y[i+1]
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  154.194 μs (0 allocations: 0 bytes)
```

MACROS COULD BE DISREGARDED OR APPLIED AUTOMATICALLY BY THE COMPILER

The influence of macros on code execution is complex. In many cases, macros might have no impact at all because compilers ultimately decide the best strategy for the problem at hand. Thus, they could already be implementing the functionality we suggest through the macro, or simply disregard it entirely. The lack of any discernible impact is easily inferred through execution times, which remain unchanged with and without the macro.

This occurs with the `@inbounds` macro, in cases where compiler is already skipping bound checks. This is only implemented automatically by the compiler in very simple cases, such as when we define iterations by `eachindex`. In such scenarios, the compiler can recognize that memory access is safe and automatically disable bounds checking, rendering the `@inbounds` macro redundant.

```julia
x = rand(1_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  3.151 μs (0 allocations: 0 bytes)
```

```julia
x = rand(1_000)

function foo(x)
    output = 0.

    @inbounds for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  3.098 μs (0 allocations: 0 bytes)
```

Macros could also serve as a mere hint to the compiler, without dictating its use. In such scenarios, the hint provided indicates that certain assumptions are met, allowing the compiler to implement more aggressive optimizations. The compiler will then carefully analyze the operations involved and decide whether the suggested approach is actually beneficial. If it is, it'll apply the optimizations. If not, it'll disregard the hint and opt for a different approach. This determines that macros guide the compiler towards better performance, but without imposing strict directives.

An example along these lines is `@simd`. This which suggests the application of SIMD instructions a technique that we'll be explored in the next sections. When `@simd` is introduced, the compiler maintains complete autonomy in deciding whether to implement the suggested optimization. In fact, it'll only adopt SIMD instructions if it concludes that they'll potentially improve performance in the specific application.

In the following example, `@simd` is ignored by the compiler, explaining why the execution time remains the same with and without the macro. [1]

```julia
x = rand(2_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = if (200_000 > i > 100_000)
                        x[i] * 1.1
                    else
                        x[i] * 1.2
                    end
    end

    return output
end
```

```
julia> @btime foo($x)
  1.056 ms (2 allocations: 15.26 MiB)
```

```julia
x = rand(2_000_000)

function foo(x)
    output = similar(x)

    @simd for i in eachindex(x)
        output[i] = if (200_000 > i > 100_000)
                        x[i] * 1.1
                    else
                        x[i] * 1.2
                    end
    end

    return output
end
```

```
julia> @btime foo($x)
  1.066 ms (2 allocations: 15.26 MiB)
```

FOOTNOTES

[1] The fact that the code implemented is the same could be confirmed by inspecting the internal code executed.