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. In this way, practitioners 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.

USES OF MACROS

Macros, denoted by the `@` symbol before their name, resemble functions in that they take input and produce output. Their primary difference lies in what they operate on and what they can return. Specifically, *functions operate on values*: they take evaluated expressions as arguments and return a final computed value. In contrast, *macros operate on code*: they take unevaluated expressions as input and return a new transformed expression, which is then compiled and executed in place of the original macro call.

This unique feature enables macros to handle tasks that functions can't. Two notable applications are worth mentioning.

First, macros can be used **to simplify code**. By automating repetitive tasks and eliminating boilerplate, macros can make code significantly more readable and maintainable. 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. This is demonstrated below.

```
x = rand(1_000)

function foo(x)
    x1 = view(x, x .> 0.7)
    x2 = view(x, x .< 0.5)
    x3 = view(x, 1:500)
    x4 = view(x, 501:1_000)

    x1, x2, x3, x4
end
```

```
x = rand(1_000)

@views function foo(x)
    x1 = x[x .> 0.7]
    x2 = x[x .< 0.5]
    x3 = x[1:500]
    x4 = x[501:1_000]

    x1, x2, x3, x4
end
```

Another application of macros is **to modify how operations are computed**, which is the focus of the current section. This lets developers package sophisticated optimization techniques, making advanced solutions accessible. As a result, users unfamiliar with a method's underlying complexity can focus on choosing the most suitable computational approach, rather than grappling with the implementation details.

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

MACROS APPLIED IN FOR-LOOPS

One distinctive feature of Julia is its ability to execute for-loops with exceptional speed. In fact, carefully optimized for-loops can reach peak 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 customize how for-loops are computed.

To illustrate this application, let's consider `@inbounds`. To understand what this macro accomplishes, we first need to understand how for-loops 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: the additional checks not only introduce computational overhead, but also limit the compiler's ability to implement certain optimizations. In situations where iterations are guaranteed to stay within an array's bounds, nonetheless, these safety checks become redundant. Consequently, we can safely boost performance by disabling bounds checking with 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.
>
> Despite this, 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.

@INBOUNDS AS AN EXAMPLE

Using a macro within a for-loop requires its inclusion 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`.

```
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)
  5.826 µs (0 allocations: 0 bytes)
```

> **Alternative Application of @inbounds**
>
> We can alternatively apply `@inbounds` individually to any specific line within the for-loop. Nonetheless, this possibility is specific to `@inbounds`. It only arises because the macro can actually be employed even outside for-loops.

```
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)
  5.938 μs (0 allocations: 0 bytes)
```

The performance advantages of `@inbounds` come not only from eliminating bounds checking itself, but also from giving the compiler more freedom to implement additional optimizations.

To understand this, note that bounds checking is essentially a conditional statement, where the iteration is executed only if all indices are within range. As we'll see in the next sections, conditional statements limit the compiler's ability to apply the so-called SIMD instructions, which are a form of parallelism within a single core.

MACROS COULD BE APPLIED AUTOMATICALLY OR DISREGARDED BY THE COMPILER

The influence of a macro on code execution isn't always predictable. In many cases, it might have no impact at all. This is because the compiler has the final say on optimization strategy. Thus, it might already be applying the optimization suggested or determine that the macro's recommendation is unhelpful and simply ignore it. In either case, you can infer a macro is ignored when there's no significant change in execution time after applying it.

Both scenarios can arise with `@inbounds` as we show below.

REDUNDANT MACROS

The compiler could prove on its own that a for-loop is safe and therefore disable bounds checking. In those cases, `@inbounds` becomes redundant. This behavior typically occurs in simple cases, such as when iterating over a single collection and using `eachindex`.

```
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)
  3.384 µs (0 allocations: 0 bytes)
```

```
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)
  3.228 µs (0 allocations: 0 bytes)
```

DISREGARDED MACRO

A macro can also be treated as a mere hint that the compiler is free to disregard. In such cases, a macro signals that the necessary conditions for a particular optimization are satisfied, allowing the compiler to consider more aggressive strategies. The compiler will then carefully analyze the operations and decide if the suggested approach is actually beneficial. This fact highlights how macros can guide the compiler toward better performance, without imposing strict directives.

A prime example along these lines is the `@simd` macro. This suggests the application of SIMD instructions, a subject that will be explored in upcoming sections. When `@simd` is added to a for-loop, the compiler retains full autonomy in deciding whether to implement the suggested optimization. In the example below, the compiler concludes that SIMD would likely degrade performance, thus ignoring `@simd`. This explains why the execution time remains the same with and without the macro.

```
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)
  881.929 μs (3 allocations: 15.259 MiB)
```

```
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)
  863.776 μs (3 allocations: 15.259 MiB)
```