8d. Type Stability with Global Variables

Martin Alfaro
PhD in Economics

INTRODUCTION

Variables can be categorized as local or global according to the code block in which they live: **global variables** can be accessed and modified anywhere in the code, while **local variables** are only accessible within a specific scope. For this section, the scope of interest is a function, so local variables will exclusively refer to function arguments and variables defined within the function body.

The distinction is especially relevant for this chapter, since **global variables are a common source of type instability**. The reason is that Julia doesn't assign concrete types to global variables. As a result, the compiler is forced to consider multiple possibilities during any computation that involves these variables. Such behavior prevents specialization, leading to reduced performance.

The current section explores two approaches to working with global variables: type-annotations and constants. Defining global variables as constants is a natural choice when values are truly fixed, such as in the case of $\pi = 3.14159$. More broadly, constants can be used in any scenario where they remain unmodified throughout the script. Compared to type annotations, constants offer better performance, as the compiler gains knowledge of *both* the type and value, rather than just the type. This feature allows for further optimizations, effectively making **the behavior of constants within a function indistinguishable from that of a literal value**. [1]

> **Warning!** - You Should Still Wrap Code in a Function
>
> Even if you implement the fixes proposed for global variables, optimal performance still calls for wrapping tasks in functions. The reason is that **functions implement additional optimizations** that aren't possible in the global scope.

WHEN ARE WE USING GLOBAL VARIABLES?

Before exploring approaches for handling global variables, let's first identify scenarios in which global variables arise. To this end, we present two cases, each represented in a different tab below. The first one considers the simplest scenario possible, where operations are performed directly in the global scope. For its part, the second one illustrates a more nuanced case, where a function accesses and operates on a global variable.

The third tab serves as a counterpoint, implementing the same operations but within a self-contained function. By definition, self-contained functions exclusively operate with locally defined variables. Thus, the comparison of the last two tabs highlights the performance lost by relying on global variables.

```
# all operations are type UNSTABLE (they're defined in the global scope)
x = 2

y = 2 * x
z = log(y)
```

```
x = 2

function foo()
    y = 2 * x
    z = log(y)

    return z
end

@code_warntype foo() # type UNSTABLE
```

```
x = 2

function foo(x)
    y = 2 * x
    z = log(y)

    return z
end

@code_warntype foo(x)  # type stable
```

Self-contained functions offer advantages that extend beyond performance gains: they **enhance readability, predictability, testability, and reusability**. These benefits were briefly considered in a previous section, and come from an interpretation of functions as embodying a specific task.

Among other benefits, self-contained functions are easier to reason about, as understanding their logic doesn't require tracking variables across the entire script. Moreover, a function's output depends solely on its input parameters, without any dependence on the script's state regarding global variables. This makes self-contained functions more predictable, additionally simplifying the code debugging process. Finally, by acting as a standalone program with a clear well-defined purpose, self-contained functions can be reapplied for similar tasks, reducing code duplication and facilitating code maintainability.

ACHIEVING TYPE STABILITY WITH GLOBAL VARIABLES

The previous subsection emphasized the benefits of self-contained functions, providing compelling reasons to avoid global variables. Nonetheless, global variables can still be highly convenient in certain scenarios. For instance, this is the case when we work with true constants. Considering this, next we present two approaches that let us work with global variables, while addressing their performance penalty.

CONSTANT GLOBAL VARIABLES

Declaring global variables as constants requires adding the `const` keyword before the variable's name, such as in `const x = 3`. This approach can be applied to variables of any type, including collections.

```
const a = 5
foo()   = 2 * a

@code_warntype foo()        # type stable
```

```
const b = [1, 2, 3]
foo()   = sum(b)

@code_warntype foo()        # type stable
```

> **Warning!** - Avoid Reassignments to Global Variables
>
> **Global variables should be declared constants only if their values will remain unchanged throughout the session**. Although it's possible to redefine constants, doing so is discouraged. The feature was only introduced to facilitate testing in interactive sessions, eliminating the need to restart Julia after each modification of a constant's value.
>
> Importantly, if you use this option, all functions depending on the constant changed must be re-declared afterwards: any function that isn't redefined will still rely on the constant's original value. Given this requirement, the safest approach is to rerun the entire script after any change.
>
> To illustrate the potential consequences of overlooking this practice, let's compare the following code snippets that execute the function `foo`. Both define a constant value of `x=1`, which is subsequently redefined as `x=2`. The first example runs the script without re-executing the definition of `foo`, in which case the value returned by `foo` is still based on `x = 1`. In contrast, the second example emulates the re-execution of the entire script. This is achieved by rerunning `foo`'s definition, thus ensuring that `foo` relies on the updated value of `x`.

```
const x = 1
foo()   = x
foo()             # it gives 1

x       = 2
foo()             # it still gives 1
```

```
const x = 1
foo()   = x
foo()             # it gives 1

x       = 2
foo()   = x
foo()             # it gives 2
```

TYPE-ANNOTATING A GLOBAL VARIABLE

The second approach to address type instability involves declaring *concrete* types for global variables. This is done by including the operator `::` after the variable's name (e.g., `x::Int64`). For vectors, note that the elements must also have a concrete type. Otherwise, type stability won't be achieved.

```
x::Int64           = 5
foo()              = 2 * x

@code_warntype foo()     # type stable
```

```
y::Vector{Float64} = [1, 2, 3]
foo()              = sum(y)

@code_warntype foo()     # type stable
```

```
z::Vector{Number}  = [1, 2, 3]
foo()              = sum(z)

@code_warntype foo()     # type UNSTABLE
```

DIFFERENCES BETWEEN APPROACHES

The two approaches presented for handling global variables have different implications for both code behavior and performance. The key lies in that **type-annotations assert a variable's type, while constants additionally declare its value**. Next, we analyze the main differences between both approaches.

DIFFERENCES IN CODE

Unlike the case of constants, type-annotations allow you to reassign a global variable without unexpected consequences. This means you don't need to re-run the entire script when redefining a variable.

```
x::Int64 = 5
foo()    = 2 * x
foo()               # output is 10

x        = 2
foo()    = 2 * x
foo()               # output is 4
```

DIFFERENCES IN PERFORMANCE

Type-annotated global variables are more flexible than constants: they require only a declaration of types, without binding to a specific value. This flexibility, however, comes at a performance cost. The reason is that constants not only convey type information, but also act as a promise of immutability throughout the program. As a result, constants behave like literal values embedded directly in code. This allows the compiler to optimize more aggressively. For example, by replacing certain expressions with their precomputed results.

The following code demonstrates this behavior. It performs an operation that can be precomputed if the value of the global variable is known at compile time. Declaring the global variable as a constant allows the compiler to replace the operation with its result, effectively treating it as a hard-coded value. In contrast, merely type-annotating the global variable constrains only its type, without fixing its value. To make the performance difference more evident, we call this operation repeatedly inside a for-loop.

```
const k1  = 2

function foo()
    for _ in 1:100_000
       2^k1
    end
end
```

```
julia> @btime foo()
  0.800 ns (0 allocations: 0 bytes)
```

```
k2::Int64 = 2

function foo()
    for _ in 1:100_000
       2^k2
    end
end
```

```
julia> @btime foo()
  115.600 µs (0 allocations: 0 bytes)
```

Invariance of Operations

Even without declaring variables as constants, the compiler could still recognize the invariance of some operations across repeated calculations. In such cases, it computes the operation once and reuses the result whenever needed.

To illustrate, consider reexpressing each element of `x` as a proportion relative to the sum of elements. A naive implementation would involve a for-loop with `sum(x)` inside the for-loop body, causing `sum(x)` to be recomputed on every iteration. By contrast, when shares are computed through `x ./ sum(x)`, the compiler is smart enough to recognize the invariance of `sum(x)` across iterations. Therefore, it proceeds to its pre-computation, eliminating redundant work.

```
x           = rand(100_000)


foo(x) = x ./ sum(x)
```

```
julia> @btime foo($x)
  49.400 µs (2 allocations: 781.30 KiB)
```

```
x           = rand(100_000)
const sum_x = sum(x)

foo(x) = x ./ sum_x
```

```
julia> @btime foo($x)
  41.500 µs (2 allocations: 781.30 KiB)
```

```
x           = rand(100_000)


function foo(x)
    y    = similar(x)

    for i in eachindex(x,y)
        y[i] = x[i] / sum(x)
    end

    return y
end
```

```
julia> @btime foo($x)
  633.245 ms (2 allocations: 781.30 KiB)
```

FOOTNOTES

[1.] Literal values refer to values expressed directly in the code (e.g., `1`, `"hello"`, or `true`), in contrast to values coming from a variable input.