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 throughout the entire codebase, while **local variables** only exist 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 frequent 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 potential types whenever these variables are used. Such behavior prevents specialization, leading to reduced performance.

The current section explores two approaches to reduce or even eliminate the detrimental effect of global variables: **type-annotations** and **constants**. Defining global variables as constants is a natural choice when values are truly constants, such as in the case of $\pi = 3.14159$. More broadly, constants are appropriate whenever a value remains unchanged throughout the program. Compared to type annotations, they 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 **constants in a function behave just like a literal value**. [1]

> **Warning!** - You Should Always 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 are unfeasible in the global scope.

WHEN ARE WE USING GLOBAL VARIABLES?

Let's begin by identifying operations that rely on global variables. To this end, we present two cases, each represented in a different tab. The first one considers the most direct use of global variables, where operations are performed directly in the global scope. The second tab illustrates a more nuanced case, where a function accesses and manipulates 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. Comparing this tab against the first two reveals the performance cost of 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 **promote clarity, predictability, testability, and reusability**. These benefits were briefly introduced in a previous section, where functions were framed as units that embody a specific task.

Specifically, self-contained functions are much easier to reason about, because all relevant information is local to the function. Due to this property, you don't need to track the state of variables scattered across the script. Instead, you can focus solely on the function's inputs and its internal logic. Self-contained functions also ensures that its behavior depends only on its arguments, not on the state of global variables. This makes its output more predictable and debugging more straightforward.Finally, a self-contained function behaves like a small, reusable program with a clearly defined purpose. Once written, it can be applied to similar tasks without modification, reducing code duplication and improving the overall maintainability of the codebase.

ACHIEVING TYPE STABILITY WITH GLOBAL VARIABLES

The advantages of self-contained functions provide strong incentives to avoid global variables. Still, there are situations where globals remain genuinely useful. A common example is when we work with true constants, defined as values that are fixed throughout the program.

With this in mind, the next section introduces two techniques that let us work with global variables, while mitigating their performance costs.

CONSTANT GLOBAL VARIABLES

Declaring a global variable as a constant simply requires prefixing its name with the `const` keyword, as in `const x = 3`. This mechanism works for any type of value, 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 as constants only if their values will remain unchanged throughout the session**. Although it's possible to redefine constants, doing so is highly 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 a constant is reassigned, every function that depends on it must be redefined as well. Otherwise, those functions will continue to use the constant's original value. Because this requirement is easy to overlook, the most reliable practice is to rerun the entire script whenever a constant is modified.
>
> To illustrate the potential consequences of ignoring this guideline, 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 x1 = 1
foo()    = x1
foo()             # it gives 1

x1       = 2

foo()             # it still gives 1
```

```
const x2 = 1
foo()    = x2
foo()             # it gives 1

x2       = 2
foo()    = x2
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 appending the operator `::` to the variable name, as in `x::Int64`. When working with vectors, ensure that their element type is also concrete. Otherwise, the variable will remain type-unstable despite the annotation.

```
x3::Int64           = 5
foo()               = 2 * x3

@code_warntype foo()     # type stable
```

```
x4::Vector{Float64} = [1, 2, 3]
foo()               = sum(x4)

@code_warntype foo()     # type stable
```

```
x5::Vector{Number}  = [1, 2, 3]
foo()               = sum(x5)

@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 fix both their types and values**. 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.

```
x6::Int64 = 5
foo()     = 2 * x6
foo()               # output is 10

x6        = 2
foo()     = 2 * x6
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 the code. ith this guarantee in place, the compiler can apply stronger optimizations. For instance, 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()
  0.791 ns (0 allocations: 0 bytes)
```

```
k2::Int64 = 2

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

```
julia> @btime foo()
  104.374 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)
  49.166 μs (3 allocations: 781.312 KiB)
```

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

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

```
julia> @btime foo($x)
  41.983 μs (3 allocations: 781.312 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)
  830.068 ms (3 allocations: 781.312 KiB)
```

FOOTNOTES

[1] A literal value is one written directly in the code (e.g., `1`, `"hello"`, or `true`), rather than provided as a value of a variable.