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. In the context of this section, the scope of interest is a function, so local variables will exclusively refer to function arguments and variables defined within the function.

The distinction between local and global variables is especially relevant for this chapter since **global variables are a common source of type instability**. The issue arises because Julia's type system doesn't assign specific concrete types to global variables. As a result, the compiler is forced to consider multiple possibilities for any computation involving these variables. This limitation 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 `π = 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!**
>
> **Global variables should only be declared constants if their value will remain unchanged throughout the script**. Although it's possible to redefine constants, this option was only introduced to facilitate testing during interactive use, thereby avoiding the need to restart a Julia session for each new constant value. Importantly, the use of this option assumes that all dependent functions are re-declared when the constant's value is modified: any function that isn't redefined will still rely on the constant's original value. This is why you should re-run the entire script **if you absolutely need to reassign the value of a constant**.
>
> 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 asserting a *concrete* type for a global variable. This is done by including the operator `::` after the variable's name (e.g., `x::Int64`).

```
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
```

Note that simply declaring a global variable with an abstract type won't resolve the type instability issue.

DIFFERENCES BETWEEN APPROACHES

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

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 the 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, as we only need to declare their types without committing to a specific value. However, this flexibility comes at the cost of performance, since they prevent certain optimizations that hold with constants. Such optimizations are feasible because constants not only provide information about their types, but also act as a promise that their value will remain fixed throughout the code. Within a function, this feature allows constants to behave like literal values embedded directly in the code. Consequently, the compiler can potentially replace certain expressions with their resulting outcome.

The following code demonstrates a scenario where this occurs. It consists of an operation that can be pre-calculated if the global variable's value is known. Thus, declaring the global variable as a constant enables the compiler to replace this operation by its result, making it equivalent to a hard-coded value. On the contrary, merely type-annotating the global variable only specializes code for the type provided. To starkly reveal the effect, we'll call this operation in 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)
```

Remark

Even without declaring a variable as a constant, the compiler could still recognize the invariance of some operations and perform optimizations accordingly. To illustrate this, suppose we want to reexpress each element of `x` as a proportion relative to the sum of the elements. A naive approach would involve a for-loop with `sum(x)` incorporated into the for-loop body, resulting in the repeated computation of `sum(x)`. If, on the contrary, we calculate shares through `x ./ sum(x)`, the compiler is smart enough to recognize the invariance of `sum(x)` across iterations, therefore proceeding to its pre-computation.

```
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)
```

```
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)
```

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.