# *8c.* Type Stability with Scalars and Vectors

[Martin Alfaro](#)

PhD in Economics

## INTRODUCTION

The previous section has defined type stability, along with approaches to checking whether the property holds. In this section, we start the analysis of type stability for specific objects. We cover in particular the case of scalars and vectors, providing practical guidance for achieving type stability with them.

## TYPES OF SCALARS AND VECTORS

Recall that the formal definition of a type-stable function is that the function's output type can be inferred from its argument types. In practice, however, we often rely on a more stringent definition, which requires that the compiler can infer a single concrete type for each expression within the function body. This property guarantees that every operation is specialized, resulting in optimal performance. Nevertheless, simply demanding that the output's type can be inferred already offers benefits, as it ensures that type instability won't be propagated when the function is called in other operations.

The principle applied to scalars is straightforward, demanding operations be performed on variables with the same concrete type (e.g., `Float64`, `Int64`, `Bool`). In contrast, type stability for vectors rather requires that the *elements* have a concrete type. The following table identifies scalars and vectors satisfying this property.

**Objects Whose Elements Have Concrete Types**

| Scalars | Vectors |
|:---:|:---:|
| `Int` | `Vector{Int}` |
| `Int64` | `Vector{Int64}` |
| `Float64` | `Vector{Float64}` |
| `Bool` | `BitVector` |

**Note**: `Int` defaults to `Int64` or `Int32`, depending on your CPU's architecture.

Next, we'll delve into type stability in scalars and vectors, considering each case separately.

# TYPE STABILITY WITH SCALARS

To make the definition of type stability for scalars operational, let's revisit some concepts about types. Recall that only concrete types like `Int64` or `Float64` can be <u>instantiated</u>, while abstract types like `Any` or `Number` can't.

Instantiation simply means that all values ultimately adopt a unique concrete type. For instance, a variable `x::Number = 2` shouldn't be interpreted as `x` having the type `Number`. Instead, it means that `x` can only be reassigned to values whose concrete type is a subtype of `Number`. Ultimately, `x` must have a concrete type, which in this case is `Int64`.

In this context, type instability may arise when operations mix `Int64` and `Float64`, although this isn't always the case. To illustrate this, we'll start showing some scenarios where mixing these types doesn't cause issues.

## TYPE PROMOTION AND CONVERSION

Julia employs various mechanisms to handle cases combining `Int64` and `Float64`. The first one is part of a concept known as **type promotion**, which converts dissimilar types to a common one whenever possible. The second one emerges when variables are type-annotated, in which case Julia engages in **type conversion**. By transforming values to the respective type declared, this feature also prevents the mix of types.

Both mechanisms are illustrated below.

```julia
foo(x,y)    = x * y

x           = 2
y           = 0.5

z               = foo(x,y)          # type stable: mixing `Int64` and `Float64` results in
`Float64`

julia> z
1.0
```

```julia
foo(x,y)    = x * y

x::Float64 = 2                  # this is converted to `2.0`
y          = 0.5

z               = foo(x,y)          # type stable: `x` and `y` are `Float64`, so predictable type
of output

julia> z
1.0
```

In the first tab, the output's type depends on the argument's types. However, in all cases the output's type can be predicted, since mixing `Int64` and `Float64` results in `Float64` due to automatic type promotion. As for the second tab, Julia transforms the value of `x` to make it consistent with the type-annotation declared. Consequently, `x * y` is computed as the product of two values with type `Float64`.

## TYPE INSTABILITY WITH SCALARS

While type promotion and conversion can handle certain situations, they certainly don't cover all cases. One such scenario is when a scalar's value depends on a conditional statement and each branch returns a value of a different type. In this situation, since the compiler only considers the types and not values, it can't determine which branch is relevant for the function call. As a result, it'll generate code that accommodates both possibilities, as it happens in the following example.

```julia
function foo(x,y)
    a = (x > y) ?  x  :  y

    [a * i for i in 1:100_000]
end

foo(1, 2)           # type stable   -> `a * i` is always `Int64`
```
```
julia> @btime foo(1,2)
  23.800 µs (2 allocations: 781.30 KiB)
```

```julia
function foo(x,y)
    a = (x > y) ?  x  :  y

    [a * i for i in 1:100_000]
end

foo(1, 2.5)         # type UNSTABLE -> `a * i` is either `Int64` or `Float64`
```
```
julia> @btime foo(1,2.5)
  43.200 µs (2 allocations: 781.30 KiB)
```

In the example, type instability will inevitably arise if `x` and `y` have different types. Note that type promotion is of no help here. The reason is that this mechanism only ensures that `a * i` will be converted to `Float64` if `a` is `Float64`, considering that `i` is `Int64`. However, the compiler also needs to consider the possibility that `a` could be `Int64`, in which case `a * i` would be `Int64`.

Given this ambiguity, the method instance created must be capable of handling both scenarios. Then, during runtime, Julia will gather more information to disambiguate the situation, and select the relevant computation implementation.

## TYPE STABILITY WITH VECTORS

Vectors in Julia are formally defined as collections of elements sharing a homogeneous type. Since operations based on vectors ultimately handle individual elements, type stability is contingent on whether the type of their elements is concrete.

In this context, it's important to distinguish between the type of the object and of its elements. This is because vectors having elements with a concrete type are themselves concrete, but elements with abstract types will still give rise to vectors with concrete types. This is clearly observed with `Vector{Any}`, a concrete type comprising elements with the abstract type `Any`.

Before the analysis of specific scenarios, we start by considering type conversion applied to vectors. This mechanism prevents the mix of types when vectors are defined.

## TYPE PROMOTION AND CONVERSION

By definition, vectors require all their elements to share the same type. This means that if you mix elements with disparate types, such as `String` and `Int64`, Julia will infer the vector's type as `Vector{Any}`. Despite this, there are cases where elements can be converted to a common type, such as when mixing `Float64` and `Int64`.

The following example shows this mechanism in an assignment, where the vector is not type annotated. In this case, all elements are converted to the most general type among the values included.

```julia
x = [1, 2, 2.5]     # automatic conversion to `Vector{Float64}`

julia> x
3-element Vector{Float64}:
 1.0
 2.0
 2.5
```

```julia
y = [1, 2.0, 3.0]     # automatic conversion to `Vector{Float64}`

julia> y
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

When assignments are instead declared with type-annotations and values are of different types, Julia will attempt to perform a conversion. If possible, this ensures that the assigned values conform to the declared type.

```
x1                        = [1,  2.0,  3.0]                        # automatic conversion to
 `Vector{Float64}`

x2::Vector{Int64}   = y1                                  # conversion to `Vector{Int64}`
```
```
julia> z2
3-element Vector{Number}:
 1.0
 2.0
 2.5
```

```
y1                        = [1,  2,  2.5]                        # automatic conversion to
 `Vector{Float64}`

y2::Vector{Number} = y1                                  # `y2` is still `Vector{Number}`
```
```
julia> z2
3-element Vector{Number}:
 1.0
 2.0
 2.5
```

```
nr_elements   = 3
z             = Vector{Any}(undef, nr_elements)     # `Vector{Any}` always

z             .= 1
```
```
julia> v
3-element Vector{Any}:
 1
 1
 1
```

## TYPE INSTABILITY

When evaluating type stability with vectors, two forms of operations must be considered. The first one involves operations that manipulate individual elements, such as `x[i]`. This scenario is analogous to the case of scalars, and therefore type stability follows the same rules.

The second scenario involves functions operating on the entire vector. In this case, type stability requires that vectors have elements with a concrete type. Note that this condition isn't sufficient to guarantee type stability, which ultimately depends on how the function implements the operation executed.

Nevertheless, packages tend to provide optimized versions of functions. Consequently, functions are typically type stable when users provide vectors with elements of a concrete type. For instance, this is illustrated below by the function `sum`, which adds all elements in a vector.

```
x1::Vector{Int}     = [1, 2, 3]

sum(x1)             # type stable
```

```
x2::Vector{Int64}    = [1, 2, 3]

sum(x2)              # type stable
```

```
x3::Vector{Float64} = [1, 2, 3]

sum(x3)              # type stable
```

```
x4::BitVector       = [true, false, true]

sum(x4)             # type stable
```

In contrast, the following vectors have elements with abstract types, which result in type instability.

```
x5::Vector{Number} = [1, 2, 3]

sum(x5)                    # type UNSTABLE -> `sum` must consider all possible subtypes of
`Number`
```

```
x6::Vector{Any}    = [1, 2, 3]

sum(x6)                 # type UNSTABLE -> `sum` must consider all possible subtypes of `Any`
```