# *8f.* Type Stability with Tuples

Martin Alfaro
PhD in Economics

## INTRODUCTION

A function is considered type stable when, given the types of its arguments, the compiler can accurately predict single concrete types for its expressions. This definition, while universal, takes on different forms when applied to specific objects. So far, we've exclusively concentrated on scalars and vectors, whose conditions for type stability are relatively straightforward.

In this section, we begin the analysis of type stability for other data structures. This is done by covering tuples. Guaranteeing type stability with tuples is more nuanced compared to vectors, as their type characterization demands more information. Its exploration will challenge our understanding of type stability, demanding a clear grasp of its definition and subtleties.

> **Warning!** - Tuples Are Only Suitable For Small Collections
>
> Remember that tuples should only be used for collections that comprise a few elements. Using them for large collections will result in significant performance degradation or directly trigger fatal errors.

## COMPARING TUPLES AND VECTORS

Tuples and vectors are the most ubiquitous forms of collections in Julia, with tuples playing a vital role for two reasons. Firstly, tuples are more performant for small objects, as they avoid the overhead of memory allocation. This feature will be expanded on when we explore *static vectors*, which are essentially tuples that can be handled as vectors. The second reason is that tuples automatically encompass the case of **named tuples**, which are merely tuples having symbols as keys instead of indices.

In comparison to vectors, tuples possess a more intricate type system. To appreciate this, let's compare the information needed for each type description.

Vectors represent collections of elements sharing a *homogeneous* type, additionally allowing for varying number of elements. Thus, the information needed to describe the types of vectors is relatively minor. For instance, a type like `Vector{Float64}` establishes that *all* elements must have type `Float64`, without any restriction on number of elements to be contained.

For their part, tuples are fixed-size collections that can accommodate *heterogeneous* types. This makes the characterization of a tuple's type more demanding, requiring both the number of elements and the type of *each* element. For instance, the variable `tup = ("hello", 1)` has type `Tuple{String,`

`Int64}`, indicating that the first element has type `String` and the second one `Int64`. Furthermore, it implicitly sets the number of elements to two, as there's no possibility of appending or removing elements.

The fact that the number of elements is part of the type becomes clear when tuples contain `N` elements of the same type `T`. For this case, Julia provides the convenient alias `Ntuple{N, Float64}`, which is just syntactic sugar for `Tuple{T, T,...,T}` where `T` appears `N` times. [1]

In the following, we show that the choice between tuples and vectors may have different implications for type stability.

## SLICES OF HETEROGENEOUS TUPLES CAN STILL BE TYPE STABLE

The type `Tuple` provides explicit information about each element's type. In contrast, vectors necessarily hold elements with a uniform type, entailing that mixing concrete types leads Julia to choose the smallest type that can encompass all of them.

In particular, vectors whose elements' types are extremely different require an abstract type to characterize all of them. In the worst case scenario, Julia could define `Vector{Any}` as the type. Defining this type propagates to slices, which will inherit the type of the parente vector. Consequenly, operations on these slices will result in type instability. Such a feature constrasts with **operations on slices of tuples, which identify a specific type for each element**.

**TUPLE**

```
tup    = (1, 2, "hello")        # type is `Tuple{Int64, Int64, String}`

foo(x) = sum(x[1:2])

@code_warntype foo(tup)         # type stable (output is `Int64`)
```

**VECTOR**

```
vector = [1, 2, "hello"]        # type is `Vector{Any}`

foo(x) = sum(x[1:2])

@code_warntype foo(vector)      # type UNSTABLE
```

Notice that type promotion could solve this issue. Through this mechanism, Julia attempts to convert each element of a vector into a common *concrete type*, thus avoiding the need of *abstract* types like `Any`. This is what occurs below, where numbers holding different types are converted to the most general concrete type.

**TUPLE**

```
tup    = (1, 2, 3.5)            # type is `Tuple{Int64, Int64, Float64}`

foo(x) = sum(x)

@code_warntype foo(tup)         # type stable (output returned is `Int64`)
```

**VECTOR**

```
vector = [1, 2, 3.5]            # type is `Vector{Float64}` (type promotion)

foo(x) = sum(x)

@code_warntype foo(vector)      # type stable (output returned is `Float64`)
```

## TUPLES CONTAIN MORE INFORMATION THAN VECTORS

Given the differences in type information, conversions between tuples and vectors can pose several challenges for type stability. To see this, let's start with the simplest case, where a tuple is converted into a vector. The conclusions drawn from this case are straightforward, as they're essentially a corollary from the previous analysis: type stability will hold when the tuple contains type-homogeneous elements or when the types are heterogeneous but can be promoted to a common type.

For the examples, recall that **each type automatically creates a function** that transforms variables into that type. In particular, below we introduce the function `Vector` with the purpose of converting variables.

**TYPE-HOMOGENEOUS TUPLES**

```
tup = (1, 2, 3)                 # `Tuple{Int64, Int64, Int64}` or just `NTuple{3, Int64}`


function foo(tup)
    x = Vector(tup)             # 'x' has type `Vector(Int64)}`
    sum(x)
end

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

**TYPE PROMOTION**

```
tup = (1, 2, 3.5)               # `Tuple{Int64, Int64, Float64}`


function foo(tup)
    x = Vector(tup)             # 'x' has type `Vector(Float64)}`
    sum(x)
end

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

**TYPE-HETEROGENEOUS TUPLES**

```
tup = (1, 2, "hello")          # `Tuple{Int64, Int64, String}`


function foo(tup)
    x = Vector(tup)            # 'x' has type `Vector{Any}`
    sum(x)
end

@code_warntype foo(tup)        # type UNSTABLE
```

For its part, **creating a tuple from a vector will inevitably cause type instability**, regardless of the vector's characteristics. The reason is that vectors don't store information about the number of elements they contain. Consequently, the compiler must treat tuples as having a variable number of arguments, with each possible number corresponding to a different concrete type.

**VECTOR WITH NON-PRIMITIVE TYPES**

```
x   = [1, 2, "hello"]          # 'Vector{Any}' has no info on each individual type


function foo(x)
    tup = Tuple(x)             # 'tup' has type `Tuple`

    sum(tup[1:2])
end

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

**VECTOR WITH PRIMITIVE TYPES**

```
x   = [1, 2, 3]                # 'Vector{Int64}' has no info on the number of elements


function foo(x)
     tup = Tuple(x)               # 'tup' has type `Tuple{Vararg{Int64}}` (`Vararg` means
"variable arguments")

    sum(tup[1:2])
end

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

## ADDRESSING VARIABLE ARGUMENTS: DISPATCH BY VALUE

A key takeaway from the previous subsection is that defining tuples from vectors invariably introduce type instability. A simple remedy for this is to convert tuples outside the function, which we then pass as function arguments. This is demonstrated in the code snippet below.

**TUPLE AS FUNCTION ARGUMENT**

```
x   = [1, 2, 3]
tup = Tuple(x)

foo(tup) = sum(tup[1:2])

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

**The approach presented should be your first option when transforming vectors to tuples**. Nonetheless, there may be scenarios where defining the tuple inside the function is unavoidable. In such cases, there are a few alternatives that can be implemented.

Note first that simply passing the vector's number of elements as a functoin argument doesn't solve the issue. The reason is that the compiler generates method instances based on information about types, not values. This means that a function argument like `length(x)` merely informs the compiler that the number of elements can be described as an object with type `Int64`, without providing any additional insight.

Instead, one effective solution is to define the tuple's length using a literal value, as demonstrated below.

**NOT A SOLUTION**

```
x   = [1, 2, 3]

function foo(x)
    tup = NTuple{length(x), eltype(x)}(x)

    sum(tup)
end

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

**INFLEXIBLE SOLUTION**

```
x   = [1, 2, 3]

function foo(x)
    tup = NTuple{3, eltype(x)}(x)

    sum(tup)
end

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

The downside of this solution is that it defeats the purpose of having generic code, as it restricts the function to tuples of a single predetermined size. To eliminate the type instability without constraining functionality, we need to introduce a more advanced solution. This is based on a technique known as **dispatch by value**. Since this approach is more complex to implement, *I recommend using it only when passing the tuple as a function argument is unfeasible*.

Next, we lay out the principles of dispatch by value, and then apply the technique to the specific case of tuples.

## DEFINITING DISPATCH BY VALUE

Dispatch by value enables passing information about values to the compiler. Implementing this feature, nonetheless, requires a workaround, since the compiler only gathers information about types. The hack consists of creating a type that stores values as type parameters. In the case of tuples, this type parameter is simply the vector's number of elements.

The functionality is implemented via the built-in type `Val`, whose use is best explained through an example. Suppose a function `foo` and a value `a` that you wish the compiler to know. The technique requires defining `foo` with a type-annotated argument having no name, `::Val{a}`. After this, you must call `foo` passing an argument `Val(a)`, which instantiates a type with parameter `a`.

To illustrate the use of `Val`, we revisit an example included in previous sections. This considers a variable `y` that could be an `Int64` or `Float64`, contingent upon a condition. The ambiguity of `y`'s type is then transmitted to any subsequent operation, leading to type instability.

Dispatch by value is implemented by defining the condition as a type parameter of `Val`. In this way, the compiler will receive information about whether condition is `true` or `false`, and therefore know `y`'s type. This makes it possible to specialize its operations.

**TYPE UNSTABLE**

```
function foo(condition)
    y = condition ? 1 : 0.5      # either `Int64` or `Float64`

    [y * i for i in 1:100]
end

@code_warntype foo(true)         # type UNSTABLE
@code_warntype foo(false)        # type UNSTABLE
```

**SOLUTION "VAL"**

```
function foo(::Val{condition}) where condition
    y = condition ? 1 : 0.5      # either `Int64` or `Float64`

    [y * i for i in 1:100]
end

@code_warntype foo(Val(true))    # type stable
@code_warntype foo(Val(false))   # type stable
```

> **Warning!**
>
> The function argument `Val` must be defined with `{}`, but called with `()`. This is because types define their parameters with `{}`, while instances of types require functions.

## DISPATCHING BY VALUE WITH TUPLES

Let's now revisit the conversion of vectors to tuples. As we previously discussed, type instability arises because vectors don't store the size as part of their type information, leaving the compiler without sufficient information to determine the tuple's type.

Dispatch by value provides a solution to this issue: by passing the vector's length as a type parameter, the function call becomes type stable.

**TYPE UNSTABLE**

```
x = [1, 2, 3]

function foo(x, N)
    tuple_x = NTuple{N, eltype(x)}(x)

    2 .+ tuple_x
end

@code_warntype foo(x, length(x))        # type UNSTABLE
```

**SOLUTION "VAL"**

```
x = [1, 2, 3]

function foo(x, ::Val{N}) where N
    tuple_x = NTuple{N, eltype(x)}(x)

    2 .+ tuple_x
end

@code_warntype foo(x, Val(length(x)))   # type stable
```

---

FOOTNOTES

[1] Don't confuse `NTuple` as an abbreviation for the type `NamedTuples`. The "N" in the former case is referring to a number "N" of elements.