8h. Gotchas for Type Stability

Martin Alfaro
PhD in Economics

INTRODUCTION

This section presents subtle scenarios where type instabilities emerge without immediate notice. Since the cause of the type instability isn't obvious, we refer to these cases as "gotchas" and offer guidance on how to address them. To ensure self-containment, we revisit some examples previously discussed, providing additional insights and recommendations for mitigation.

GOTCHA 1: INTEGERS AND FLOATS

It's essential to remember that `Int64` and `Float64` are distinct types. Although Julia could potentially convert types and mitigate the issue, mixing these types can still inadvertently introduce type instability.

To illustrate this, consider a function `foo` that takes a numeric variable `x` as its argument and performs two tasks. First, it defines a variable `y` by transforming `x` in a way that all negative values are replaced with zero. Second, it executes an operation based on the resulting `y`.

The following example illustrates two implementations of `foo`. The first one suffers from type instability, while the second one addresses the issue.

```
function foo(x)
    y = (x < 0) ?  0  :  x

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

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type UNSTABLE
```

```
function foo(x)
    y = (x < 0) ?  zero(x)  :  x

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

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type stable
```

The first implementation uses the literal `0`, which has type `Int64`. If `x` is also `Int64`, no type instability arises. However, if `x` is `Float64`, the compiler must consider that `y` could be either `Int64` or `Float64`, thus causing type instability. [1]

Julia can handle combinations of `Int64` and `Float64` quite effectively. Therefore, the latter type instability wouldn't be a significant issue if the operation involving `y` calls `y` only once. Indeed, `@code_warntype` would only issue a yellow warning that therefore could be safely ignored. However, `foo` in our example repeatedly performs an operation that involves `y`, incurring the cost of type instability multiple times. As a result, `@code_warntype` issues a red warning, indicating a more serious performance issue.

The second tab proposes a **solution** based on a function that returns the zero element corresponding to the type of `x`. This approach can be extended to other values by using either the function `convert(typeof(x), <value>)` or `oftype(x, <value>)`. Both convert `<value>` to the same type as `x`. For instance, below we reimplement `foo`, but using the value `5` instead of `0`.

```
function foo(x)
    y = (x < 0) ?  5  :  x

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

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type UNSTABLE
```

```
function foo(x)
    y = (x < 0) ?  convert(typeof(x), 5)  :  x

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

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type stable
```

```
function foo(x)
    y = (x < 0) ?  oftype(x, 5)  :  x

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

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type stable
```

GOTCHA 2: COLLECTIONS OF COLLECTIONS

In data analysis, it's common to work with collections of collections, where one structure contains others nested inside. A familiar example is the `DataFrames` package in Julia, which organizes data into columns representing different variables. Since we haven't introduced this package, we'll consider a simpler, but analogous case: a vector of vectors, whose type is `Vector{Vector}`.

The appeal of `Vector{Vector}` lies in its flexibility. Because the type doesn't constrain the contents of its inner vectors, it can represent heterogeneous data. Thus, we can create datasets that mix strings, floating-point numbers, and integers across columns.

However, this flexibility introduces a drawback. The type system only knows that each element is a vector, without knowing the concrete type of its contents. As a result, the compiler can't infer types when operating on these inner vectors, leading to type instability.

To make the issue more concrete, consider a vector `data` that contains several inner vectors. Suppose we define a function `foo` that takes `data` as its argument and performs some operation on one of its inner vectors, say `vec2`. The first tab below shows the compiler only knows that `vec2` is a vector, but it can't determine the concrete type of its elements. As a result, calls to `foo` suffer from type instability.

A straightforward way to **address** this problem is presented in the second tab. The solution consists of introducing a barrier function that takes the inner vector `vec2` as its argument. The barrier function rectifies the type instability by attempting to identify a concrete type for `vec2`.

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

function foo(data)
    for i in eachindex(data[2])
        data[2][i] = 2 * i
    end
end

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

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

foo(data) = operation!(data[2])

function operation!(x)
    for i in eachindex(x)
        x[i] = 2 * i
    end
end

@code_warntype foo(data)            # barrier-function solution
```

Note that the second tab defines the barrier function in-place. This means that the function directly modifies the contents of the inner vector `vec2`, rather than creating a new copy. Consequently, the outer structure `data` is updated as well. This in-place strategy is common in data analysis, where the goal is often to transform a dataset instead of generating a new one each time its values are modified.

GOTCHA 3: BARRIER FUNCTIONS

Barrier functions are an effective technique to mitigate type instabilities. However, it's essential to remember that **the parent function may remain type unstable**. When this occurs and instability isn't resolved before executing a repeated operation, the associated performance penalty will be incurred multiple times.

To illustrate this point, let's revisit the last example involving a vector of vectors. Below, we present two incorrect approaches to using a barrier function, followed by a demonstration of its proper application.

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

operation(i) = (2 * i)

function foo(data)
    for i in eachindex(data[2])
        data[2][i] = operation(i)
    end
end

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

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

operation!(x,i) = (x[i] = 2 * i)

function foo(data)
    for i in eachindex(data[2])
        operation!(data[2], i)
    end
end

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

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

function operation!(x)
    for i in eachindex(x)
        x[i] = 2 * i
    end
end

foo(data) = operation!(data[2])

@code_warntype foo(data)            # barrier-function solution
```

GOTCHA 4: INFERENCE IS BY TYPE, NOT BY VALUE

Julia's compiler generates method instances solely on the basis of types, without taking actual values into account. To demonstrate this, let's consider the following example.

```
function foo(condition)
    y = condition ?  2.5  :  1

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

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

At first glance, we might erroneously conclude that `foo(true)` is type stable: the value of `condition` is `true`, so that `y = 2.5` and therefore `y` will have type `Float64`. However, values don't participate in multiple dispatch, meaning that Julia's compiler ignores the value of `condition` when inferring `y`'s type. Ultimately, `y` is treated as potentially being either `Int64` or `Float64`, leading to type instability.

The issue in this case can be easily resolved by replacing `1` by `1.0`, thus ensuring that `y` is always `Float64`. More generally, we could employ similar techniques to the first "gotcha", where values are converted to a specific concrete type.

An alternative solution relies on dispatching by value, a technique we already explored and implemented for tuples. This technique makes it possible to pass information about values to the compiler. It's based on the type `Val`, along with the keyword `where` introduced here.

Specifically, for any function `foo` and value `a` that you seek the compiler to know, you need to include `::Val{a}` as an argument. In this way, `a` is interpreted as a type parameter, which you can identify by including the keyword `where`. Finally, we need to call `foo` by passing `Val(a)` as its input.

Applied to our example, type instability in `foo` emerges because the value of `condition` isn't known by the compiler. Dispatching by `a` enables us to pass the value of `condition` to the compiler.

```
function foo(condition)
    y = condition ?  2.5  :  1

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

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

```
function foo(::Val{condition}) where condition
    y = condition ?  2.5  :  1

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

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

GOTCHA 5: VARIABLES AS DEFAULT VALUES OF KEYWORD ARGUMENTS

Functions accept positional and keyword arguments. Moreover, when functions are particularly defined with keyword arguments, it's also possible to assign default values. However, if these default values are set through variables rather than literal values, a type instability will be introduced. The reason is that these variables will be treated as global variables.

```
foo(; x) = x

β = 1
@code_warntype foo(x=β)          #type stable
```

```
foo(; x = 1) = x


@code_warntype foo()            #type stable
```

```
foo(; x = β) = x

β = 1
@code_warntype foo()            #type UNSTABLE
```

When setting a variable as a default value is unavoidable, there are still a few strategies you could follow to restore type stability.

One set of solutions leverages the techniques we introduced for global variables. hese include type-annotating the global variable (*Solution 1a*) or defining it as a constant (*Solution 1b*).

Another strategy involves defining a function that stores the default value. By doing so, you can take advantage of type inference, where the function attempts to infer a concrete type for the default value (*Solution 2*).

You can also adopt a local approach, by adding type annotations to either the keyword argument (*Solution 3a*) or the default value itself (*Solution 3b*). Finally, type instability does not arise when positional arguments are used as default values of keyword arguments (*Solution 4*).

All these cases are stated below.

```
foo(; x = β) = x

const β = 1
@code_warntype foo()            #type stable
```

```
foo(; x = β) = x

β::Int64 = 1
@code_warntype foo()            #type stable
```

```
foo(; x = β()) = x

β() = 1
@code_warntype foo()            #type stable
```

```
foo(; x::Int64 = β) = x

β = 1
@code_warntype foo()            #type stable
```

```
foo(; x = β::Int64) = x

β = 1
@code_warntype foo()            #type stable
```

```
foo(β; x = β) = x

β = 1
@code_warntype foo(β)           #type stable
```

GOTCHA 6: CLOSURES CAN EASILY INTRODUCE TYPE INSTABILITIES

Closures are a fundamental concept in programming. They refer to functions that capture and retain access to variables from the scope in which they were defined. In practical terms, a closure arises when **one function is defined inside another**, including the case where anonymous functions are used inside a function.

Although closures provide a convenient way to write modular and self-contained code, they can sometimes introduce type instabilities. While Julia has made progress in mitigating these issues, they have persisted for years and remain a source of potential inefficiency. For this reason, it's essential to understand not only the consequences of using closures carelessly, but also to learn strategies for addressing their performance challenges.

CLOSURES ARE COMMON IN CODING

There are several scenarios where closures emerge naturally. One such scenario is when a task requires multiple steps, but you prefer to keep a single self-contained unit of code. For instance, this approach is particularly useful if a function needs to perform multiple interdependent steps, such as data preparation (e.g., setting parameters or initializing variables) and subsequent computations based on that data. By nesting a function within another, you can keep related code organized and contained within the same logical block, promoting code readability and maintainability.

To illustrate how code implements a task with and without closures, we'll use generic code. This isn't intended to be executed, but rather to demonstrate the underlying structure.

```
function task()
        # <here, you define parameters and initialize variables>

    function output()
        # <here, you do some computations with the variables and parameters>
    end

    return output()
end

task()
```

```
function task()
        # <here, you define parameters and initialize variables>

    return output(<variables>, <parameters>)
end

function output(<variables>, <parameters>)
        # <here, you do some computations with the variables and parameters>
end

task()
```

Although the approach using closures may seem more intuitive, it can easily introduce type instability. This occurs when one of these conditions hold:

- variables or arguments are redefined inside the function (e.g., when updating a variable)
- the order in which functions are defined is altered
- anonymous functions are introduced

Each of these cases is explored below, where we refer to the containing function as the *outer function* and the closure as the *inner function*.

WHEN THE ISSUE ARISES

Let's start examining three examples. They cover all the possible situations where closures could result in type instability, allowing us to identify real-world scenarios.

The first examples reveal that the placement of the inner function could matter for type stability.

```
function foo()
    x           = 1
    bar()       = x

    return bar()
end

@code_warntype foo()      # type stable
```

```
function foo()
    bar(x)      = x
    x           = 1

    return bar(x)
end

@code_warntype foo()      # type stable
```

```
function foo()
    bar()        = x
    x            = 1

    return bar()
end

@code_warntype foo()      # type UNSTABLE
```

```
function foo()
    bar()::Int64 = x::Int64
    x::Int64     = 1

    return bar()
end

@code_warntype foo()      # type UNSTABLE
```

```
function foo()
    x = 1

    return bar(x)
end

bar(x) = x

@code_warntype foo()      # type stable
```

The second example establishes that type instability arises when closures are combined with reassignments of variables or arguments. This issue even emerges when the reassignment involves the same object, including trivial expressions such as `x = x`. The example also reveals that type annotating the redefined variable or the closure doesn't resolve the problem.

```
function foo()
    x            = 1
    x            = 1            # or 'x = x', or 'x = 2'

    return x
end

@code_warntype foo()            # type stable
```

```
function foo()
    x            = 1
    x            = 1            # or 'x = x', or 'x = 2'
    bar(x)       = x

    return bar(x)
end

@code_warntype foo()            # type stable
```

```
function foo()
    x            = 1
    x            = 1            # or 'x = x', or 'x = 2'
    bar()        = x

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```
function foo()
    x::Int64     = 1
    x            = 1
    bar()::Int64 = x::Int64

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```
function foo()
    x::Int64     = 1
    bar()::Int64 = x::Int64
    x            = 1

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```
function foo()
    bar()::Int64 = x::Int64
    x::Int64     = 1
    x            = 1

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```
function foo()
    x            = 1
    x            = 1            # or 'x = x', or 'x = 2'

    return bar(x)
end

bar(x) = x

@code_warntype foo()            # type stable
```

Finally, the last example deals with situations involving multiple closures. It highlights that the order in which you define them could matter for type stability. The third tab in particular demonstrates that passing subsequent closures as arguments can sidestep the issue. However, such an approach is at odds with how code is generally written in Julia.

```
function foo(x)
    closure1(x) = x
    closure2(x) = closure1(x)

    return closure2(x)
end

@code_warntype foo(1)            # type stable
```

```
function foo(x)
    closure2(x) = closure1(x)
    closure1(x) = x

    return closure2(x)
end

@code_warntype foo(1)            # type UNSTABLE
```

```
function foo(x)
    closure2(x, closure1) = closure1(x)
    closure1(x)           = x

    return closure2(x, closure1)
end

@code_warntype foo(1)            # type stable
```

```
function foo(x)
    closure2(x) = closure1(x)

    return closure2(x)
end

closure1(x) = x

@code_warntype foo(1)            # type stable
```

In the following, we'll examine specific scenarios where these patterns emerge. The examples reveal that the issue can occur more frequently than we might expect. For each scenario, we'll also provide a solution that enables the use of a closure approach. Nonetheless, if the function captures a performance critical part of your code, it's probably wise to avoid closures.

"BUT NO ONE WRITES CODE LIKE THAT"

i) Transforming Variables through Conditionals

```
x = [1,2]; β = 1

function foo(x, β)
    (β < 0) && (β = -β)          # transform 'β' to use its absolute value

    bar(x) = x * β

    return bar(x)
end

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

```
x = [1,2]; β = 1

function foo(x, β)
    (β < 0) && (β = -β)          # transform 'β' to use its absolute value

    bar(x,β) = x * β

    return bar(x,β)
end

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

```
x = [1,2]; β = 1

function foo(x, β)
    δ = (β < 0) ? -β : β        # transform 'β' to use its absolute value

    bar(x) = x * δ

    return bar(x)
end

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

```
x = [1,2]; β = 1

function foo(x, β)
     β = abs(β)                      # 'δ = abs(β)' is preferable (you should avoid redefining
variables)

    bar(x) = x * δ

    return bar(x)
end

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

Recall that the compiler doesn't dispatch by value, and so whether the condition holds is irrelevant. For instance, the type instability would still hold if we wrote `1 < 0` instead of `β < 0`. Moreover, the value used to redefine `β` is also unimportant, with the same conclusion holding if you write `β = β`.

ii) Anonymous Functions inside a Function

Using an anonymous function inside a function is another common form of closure. Considering this, type instability also arises in the example above if we replace the inner function `bar` for an anonymous function. To demonstrate this, we apply `filter` with an anonymous function that keeps all the values in `x` that are greater than `β`.

```
x = [1,2]; β = 1

function foo(x, β)
    (β < 0) && (β = -β)         # transform 'β' to use its absolute value

    filter(x -> x > β, x)       # keep elements greater than 'β'
end

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

```
x = [1,2]; β = 1

function foo(x, β)
    δ = (β < 0) ? -β : β        # define 'δ' as the absolute value of 'β'

    filter(x -> x > δ, x)       # keep elements greater than 'δ'
end

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

```
x = [1,2]; β = 1

function foo(x, β)
     β = abs(β)                        # 'δ = abs(β)' is preferable (you should avoid redefining
variables)

    filter(x -> x > β, x)       # keep elements greater than β
end

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

iii) Variable Updates

```
function foo(x)
    β = 0                       # or 'β::Int64 = 0'
    for i in 1:10
        β = β + i               # equivalent to 'β += i'
    end

    bar() = x + β               # or 'bar(x) = x + β'

    return bar()
end

@code_warntype foo(1)           # type UNSTABLE
```

```
function foo(x)
    β = 0
    for i in 1:10
        β = β + i
    end

    bar(x,β) = x + β

    return bar(x,β)
end

@code_warntype foo(1)           # type stable
```

```
x = [1,2]; β = 1

function foo(x, β)
    (1 < 0) && (β = β)

    bar(x) = x * β

    return bar(x)
end

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

iv) The Order in Which you Define Functions Could Matter Inside a Function

To illustrate this claim, suppose you want to define a variable `x` that depends on a parameter `β`. However, `β` is measured in one unit (e.g., meters), while `x` requires `β` to be expressed in a different unit (e.g., centimeters). This implies that, before defining `x`, we must rescale `β` to the appropriate unit.

Depending on how we implement the operation, a type instability could emerge.

```
function foo(β)
    x(β)                  =  2 * rescale_parameter(β)
    rescale_parameter(β)  =  β / 10

    return x(β)
end

@code_warntype foo(1)      # type UNSTABLE
```

```
function foo(β)
    rescale_parameter(β)  =  β / 10
    x(β)                  =  2 * rescale_parameter(β)

    return x(β)
end

@code_warntype foo(1)      # type stable
```

FOOTNOTES

[1] A similar problem would occur if we replaced `0` by `0.` and `x` is an integer.