8g. Type Stability with Higher-Order Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

Functions in Julia are **first-class objects**, a concept also referred to as **first-class citizens**. This means that functions can be handled just like any other variable: we can define vectors of functions, have functions whose outputs are other functions, and do many more sophisticated things that would be impossible if functions were treated as different entities.

In particular, the property makes it possible to define **higher-order functions**, which are functions that take another function as an argument. We've already worked with several of them, often in the form of anonymous functions passed as function arguments. A familiar example is `map(<function>, <collection>)`, which applies `<function>` to every element of `<collection>`.

In this section, the focus will be on conditions under which higher-order functions are type-stable. As we'll discover, these functions present some challenges in this regard.

> **Remark**
>
> Throughout the explanations, we'll often refer to the function passed as an argument as the *callback function*.

THE ISSUE

In Julia, *each function defines its own unique concrete type*. In turn, this concrete type is a subtype of an abstract type called `Function`. The type `Function` encompasses all possible functions defined in Julia. The design of the type system creates challenges when specializing the computation method of higher-order functions. Specifically, it can potentially lead to a combinatorial explosion of methods, where a unique method is generated for each callback function.

To address this issue, Julia takes a conservative stance, **often choosing not to specialize the methods of high-order functions**. In particular, we'll see that Julia avoids specialization if the callback function isn't explicitly called. The performance in those cases can drop sharply, as the execution runtime would become similar to performing operations in the global scope.

Given this, it's important to pinpoint the scenarios where specialization is inhibited and monitor its consequences. If you notice that performance is severely impaired, there are still ways to enforce specialization. In this section, we'll explore these strategies.

AN EXAMPLE OF NO SPECIALIZATION

Let's illustrate the conditions under which higher-order functions fail to specialize. Consider a scenario where the goal is to sum the transformed elements of a vector `x`. The only requirement imposed is that the transforming function should be generic, allowing us to possibly apply different functions for the transformation.

We implement this construction via a higher-order function `foo`. The function applies broadcasting to transform `x` through some function `f`. To demonstrate how `foo` works, we call it with the function `abs` as the transformation function, which provides absolute values.

```
x         = rand(100)

foo(f, x) = f.(x)
```

```
julia> @code_warntype foo(abs,x)
```

Even when `foo(abs,x)` isn't specialized, `@code_warntype` **fails to detect any type-stability issues**. This is a consequence of `@code_warntype` evaluating type stability *under the assumption that specialization is attempted*. In our example, this assumption doesn't hold and therefore `@code_warntype` is of no use.

Type instability in this case arises because Julia **avoids specialization if a callback function isn't explicitly called within the function**. In the example, the function `f` only enters `foo` as an argument of broadcasting, but there's no explicit line calling `f`.

To obtain indirect evidence about the lack of specialization, we can compare the execution times of the original `foo` function with a version that explicitly calls `f`.

```
x = rand(100)

function foo(f, x)

    f.(x)
end
```

```
julia> foo(abs, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo(abs, $x)
  817.824 ns (12 allocations: 1.250 KiB)
```

```
x = rand(100)

function foo(f, x)
    f(1)                # irrelevant computation to force specialization
    f.(x)
end
```

```
julia> foo(abs, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo(abs, $x)
  26.099 ns (2 allocations: 928 bytes)
```

The comparison reveals a significant reduction in execution time when `f(1)` is added, along with a notable decrease in memory allocations. As we'll demonstrate in future sections, excessive allocations are often indicative of type instability.

FORCING SPECIALIZATION

> ⚠ **Warning!**
>
> Exercise caution when forcing specialization. Overly aggressive specialization can degrade performance severely, explaining why Julia's default approach is deliberately conservative. In particular, you should avoid specialization when your script repeatedly calls a high-order function with many unique functions.

Explicitly calling the callback function to circumvent the no-specialization issue isn't ideal, as it introduces an unnecessary computation. Fortunately, alternative solutions exist. One of them is to type-annotate `f`, which provides Julia with a hint to specialize the code for that type of function.

Another solution involves wrapping the function in a tuple before passing it as an argument. This ensures the identification of the function's type, as tuples define a concrete type for each of their elements.

Below, we outline both approaches.

```
x = rand(100)

function foo(f, x)

    f.(x)
end
```

```
julia> foo(abs, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo(abs, $x)
  817.824 ns (12 allocations: 1.250 KiB)
```

```
x = rand(100)


function foo(f::F, x) where F
    f.(x)
end
```

```
julia> foo(abs, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo(abs, $x)
  27.302 ns (2 allocations: 928 bytes)
```

```
x = rand(100)
f_tup = (abs,)

function foo(f_tup, x)
    f_tup[1].(x)
end
```

```
julia> foo(f_tup, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo($f_tup, $x)
  28.238 ns (2 allocations: 928 bytes)
```