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 treated like any other variable, thus allowing for vectors of functions, functions returning other functions, and more.

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 encountered several examples of higher-order functions, often in the form of anonymous functions passed as arguments. A classic example of a higher-order function is `map(<function>, <collection>)`, given that `<function>` is one of its arguments. Throughout the explanations, we follow common terminology and refer the function passed as an argument as the *callback function*.

In this section, we'll examine conditions under which higher-order functions are type-stable. As we'll see, these functions present some challenges for specializing their computation method.

FUNCTIONS AS ARGUMENTS: THE ISSUE

For exploring type stability, a distinctive feature of functions is that *each function defines its own unique concrete type*. In turn, this concrete type is a subtype of an abstract type called `Function`, which encompasses all possible functions defined in Julia. This type system creates challenges when specializing the computation method of higher-order functions, as it can potentially lead to a combinatorial explosion of methods, with a unique method generated for each callback function.

To address this issue, Julia takes a conservative approach, **often choosing not to specialize the methods of high-order functions**. The performance in such cases can be severely degraded, as the execution runtime would become similar to performing operations in the global scope.

Taking this into account, it's important to pinpoint the scenarios where specialization is inhibited and monitor its consequences. If it occurs that performance is severely impaired, there are still ways to enforce specialization. In the following section, we explore several techniques for doing so.

> **Warning!**
>
> Exercise caution when inducing 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. [1]

AN EXAMPLE OF NO SPECIALIZATION

To illustrate when higher-order functions don't specialize, let's consider a specific scenario where we sum the transformed elements of a vector `x`. The only requirement we impose is that the transforming function should be generic, allowing us to possibly apply different functions for the transformation.

Below, we implement this operation through a higher-order function `foo`. This first uses `map` to transform `x` through some function `f`, and then applies the function `sum` to add the transformed elements. For demonstrating how `foo` works, we also call it with the function `abs` as its specific transformed function.

```
x = rand(100)

function foo(f, x)
    y = map(f, x)

    sum(y)
end
```

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

Although we can show that `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.

The specific reason for this behavior is that Julia **avoids specialization when a callback function isn't explicitly called within the function**. In the example, the function `f` only enters `foo` as an argument of `map`, but there's no explicit line calling `f`.

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

```
x = rand(100)

function foo(f, x)
    y = map(f, x)


    sum(y)
end
```

```
julia> foo(abs, x)
48.447
julia> @btime foo(abs, $x)
  195.579 ns (3 allocations: 928 bytes)
```

```
x = rand(100)

function foo(f, x)
    y = map(f, x)
    f(1)                # irrelevant computation to force specialization

    sum(y)
end
```

```
julia> foo(abs, x)
48.447
julia> @btime foo(abs, $x)
  45.745 ns (1 allocation: 896 bytes)
```

The comparison lays bare a significant reduction in time when `f(1)` is added. Furthermore, there's also a notable decrease in memory allocations. As we'll demonstrate when exploring the subject, excessive allocations often serve as a telltale sign of type instability.

FORCING SPECIALIZATION

Explicitly calling the function to circumvent the no-specialization issue isn't optimal, since it introduces an unnecessary computation. Fortunately, alternative solutions exist to address the problem. One approach is to type-annotate `f`, which provides Julia with a hint to specialize. Another solution is to wrap the function in a tuple and then call it. This ensures the identification of the function's type, as tuples identify a concrete type for each element.

Below, we outline both approaches.

```
x = rand(100)


function foo(f::F, x) where F
    y = map(f, x)


    sum(y)
end
```

```
julia> foo(abs, x)
48.447
julia> @btime foo(abs, $x)
  46.686 ns (1 allocation: 896 bytes)
```

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

function foo(f_tup, x)
    y = map(f_tup[1], x)


    sum(y)
end
```

```
julia> foo(f_tup, x)
48.447
julia> @btime foo($f_tup, $x)
  45.101 ns (1 allocation: 896 bytes)
```

FOOTNOTES

[1] For discussions about the issue of excessive specialization, see here and here.