8e. Barrier Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

This section presents an approach to mitigating type instability based on the so-called **barrier functions**. These are defined as type-stable functions embedded within a type-unstable function, where variables of uncertain type are passed as arguments. This design prompts the compiler to infer concrete types for those variables, effectively creating a "barrier" that prevents the propagation of type instability to subsequent operations.

A key benefit of this technique is that **barrier functions are agnostic to the underlying source of type instability**, making them widely applicable across scenarios.

> **Warning!** - Barrier Functions Should Be Considered as a Second Option
>
> Barrier functions are preferred for situations where type instability is either difficult to fix or inherent to the operations performed. Keep in mind that the original function will remain type unstable, entailing different consequences depending on the instability nature. For this reason, it's best to aim for type-stable code from the outset, whenever possible.

APPLYING BARRIER FUNCTIONS

To illustrate the technique, let's revisit a type-unstable function from a previous section. This defines a variable `y` based on `x`, and subsequently performs an operation involving `y`.

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

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

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

In the example, `0` is an `Int64`, whereas `x` could be either an `Int64` or `Float64`. This leads to two possibilities:

- `x` is an `Int64`: then, `y` will also be an `Int64`, making `foo(1)` type stable.

- `x` is a `Float64`: the compiler then can't determine whether `y` will be an `Int64` or a `Float64`, rendering `foo(1.)` type unstable.

A barrier function can address the type instability of the second case. It requires embedding a type-stable function into `foo`, passing `y` as an argument. The function will then attempt to deduce `y`'s type, allowing the compiler to use this information for subsequent operations. The example below defines `operation` as a barrier function. [1]

```
operation(y) = [y * i for i in 1:100]

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

    operation(y)
end

@code_warntype operation(1)    # barrier function is type stable
@code_warntype operation(1.)   # barrier function is type stable

@code_warntype foo(1)          # type stable
@code_warntype foo(1.)         # barrier-function solution
```

With the introduction of `operation`, the variable `y` in `foo(1.)` can still be either an `Int64` or a `Float64`. Nevertheless, this ambiguity no longer matters, as `operation(y)` will determine the type of `y` before the array comprehension is executed. As a result, the expression `[y * i for i in 1:100]` will be computed with a method specialized for the specific type of `y`, ensuring type stability.

> **Warning!**
>
> Barrier Functions should address the type instability *before* the type-unstable operation is executed. Otherwise, we're back to the original issue, where the compiler has to check `y`'s type at each iteration and select a method accordingly.
>
> For example, `foo` in the example below doesn't apply the technique correctly: `y` can be either `Float64` or `Int64`, but `operation(y,i)` only identifies the type inside the for-loop. Thus, the compiler is forced to check `y`'s type at each iteration, which is the original problem we intended to solve.

```
operation(y,i) = y * i

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

    [operation(y,i) for i in 1:100]
end

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

REMARKS ON @CODE_WARNTYPE

Functions introducing barrier functions hinder the interpretation of `@code_warntype`. This is because barrier functions typically mitigate type instability, rather than completely eliminating it. And even if the barrier function successfully eliminates the type instability, a red warning may still be triggered.

To illustrate this, let's start presenting a scenario where the barrier function completely eliminates the type instability. Yet, a red warning shows up.

```
x = ["a", 1]                    # variable with type 'Any'



function foo(x)
    y = x[2]

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

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

```
x = ["a", 1]                    # variable with type 'Any'

operation(y) = [y * i for i in 1:100]

function foo(x)
    y = x[2]

    operation(y)
end
```

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

In this example, `y` is defined from an object with type `Vector{Any}`. This leads to a red warning, as `x[2]` has type `Any` and therefore the compiler can't infer a concrete type for `y`. However, no operation is involved at that point, as we're only performing an assignment. Since the only operation performed uses a barrier function, the lack of type information is inconsequential. Overall, type instability is never impacting performance after introducing a barrier function.

In contrast, the example below demonstrates that a barrier function may only alleviate type instability, rather than eliminate it entirely. In this scenario, the operation `2 * x[2]` is type unstable, forcing the compiler to generate code for each possible concrete type of `x[2]`. Nonetheless, this operation has a negligible performance impact on `foo`, justifying why the barrier function only targets the more demanding operation.

```
x = ["a", 1]                    # variable with type 'Any'



function foo(x)
    y = 2 * x[2]

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

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

```
x = ["a", 1]                    # variable with type 'Any'

operation(y) = [y * i for i in 1:100]

function foo(x)
    y = 2 * x[2]

    operation(y)
end
```

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

```
x = ["a", 1]                    # variable with type 'Any'

operation(y) = [y * i for i in 1:100]

function foo(z)
    y = 2 * z

    operation(y)
end
```

```
julia> @code_warntype foo(x[2])
```

The effectiveness of a barrier function ultimately hinges on how the function `foo` will be applied. In the given example, the barrier-function solution would be sufficient if `foo` is called only once. Instead, if `foo` is eventually called in a tight loop, the type instability of `2 * x[2]` would be incurred multiple times. In such cases, simultaneously addressing the type instability in `2 * x[2]` could lead to substantial performance benefits.

FOOTNOTES

[1] In this particular example, there's an easier solution, where `0` is substituted with `zero(x)`. The function `zero(x)` has been designed to return the additive identity (i.e., the null element) of `x`'s type.