8b. Defining Type Stability

Martin Alfaro
PhD in Economics

INTRODUCTION

This section formally defines type stability and reviews the tools employed for its verification. In the next section, we'll begin examining how type stability applies in specific scenarios.

AN INTUITION

In a previous section, we described the process that unfolds when a function is called. To briefly review, let's consider a function `foo(x) = x + 2` and executing `foo(a)` for some variable `a`. We assume that `a` has a specific value assigned and therefore a concrete type, although we omit explicitly stating values for `a`. In this way, we highlight that the process depends on types, rather than values.

Calling `foo(a)` prompts Julia to identify the concrete type of `a`, which we'll denote as `T`. If a compiled method instance for `foo` with an argument of type `T` already exists, then `foo(a)` is executed immediately. Otherwise, Julia compiles a method instance for evaluating `a + 2`. This code generation leverages type inference, wherein the compiler attempts to deduce concrete types for all involved terms. The resulting machine code is then stored (cached), making it readily available for subsequent calls of `foo(b)` when `b` has type `T`.

TYPE STABILITY AND PERFORMANCE

The key to generating fast code lies in the information available to the compiler during the compilation stage. This information is primarily gathered through type inference, where Julia identifies the specific type of each variable and expression involved. When the compiler can **accurately predict a single concrete type for the function's output**, the function call is said to be **type stable**.

While this constitutes the formal definition of type stability, a more stringent definition is usually applied in practice: the compiler must be able to **infer unique concrete types for *each* expression within the function**, not only for the final output. This definition aligns with `@code_warntype`, the built-in macro to detect type instabilities.

If the condition is satisfied, the compiler can specialize the computational approach for each operation, resulting in fast execution. Essentially, type stability dictates that there's sufficient information to determine a straight execution path, thus avoiding unnecessary type checks and dispatches at runtime.

In contrast, type-unstable functions generate generic code that accommodates each possible combination of unique concrete types. This results in additional overhead during runtime, where Julia is forced to dynamically gather type information and perform extra calculations based on it. The consequence is a pronounced deterioration in performance.

> **Type Stability Characterizes Function *Calls***
>
> It's common to describe a function as "type stable". Nevertheless, it's not the function itself that's type stable, but rather the function calls for specific concrete types of its arguments. The distinction is crucial in practice, since a function may exhibit type stability for certain input types but not others.

AN EXAMPLE

To see type stability in practice, let's consider the following example.

```
x = [1, 2, 3]                   # `x` has type `Vector{Int64}`

@btime sum($x[1:2])             # type stable
```

```
  22.406 ns (1 allocation: 80 bytes)
```

```
x = [1, 2, "hello"]             # `x` has type `Vector{Any}`

@btime sum($x[1:2])             # type UNSTABLE
```

```
  31.938 ns (1 allocation: 64 bytes)
```

The two operations may seem equivalent, as they both ultimately compute `1 + 2`. However, the methods used in each case differ, with the first approach being faster because the function call is type stable.

Specifically, the output `x[1] + x[2]` in the first tab can be deduced to be `Int64`, thus satisfying the definition of type stability. This occurs because `x[1]` and `x[2]` can be identified as `Int64`, allowing the compiler to generate code specialized for this type. Note that the efficiency of the generated code isn't limited to the given operation: it applies to any call `sum(y)` such that `y` is a `Vector{Float64}`.

In contrast, **the second tab defines a type-unstable function call**. Since `x` has type `Vector{Any}`, it becomes impossible to predict a unique concrete type for `x[1] + x[2]` solely based on `x`'s type. This is because `x[1]` and `x[2]` may embody any concrete type that is a subtype of `Any`. Consequently, the compiler is forced to create code with multiple conditional statements, with each branch handling how to compute `x[1] + x[2]` for a possible type (`Int64`, `Float64`, `Float32`, etc.). This results in slow compiled code, as it'll require extra work during runtime. Furthermore, the degraded performance will be incurred for every call `sum(y)` such that `y` has type `Vector{Any}`.

> **Remark**
>
> Julia's developers are continually refining the compiler, addressing and mitigating the effects of certain type instabilities. As a result, **many operations that were once type unstable are now type stable**. This means that type stability should be considered a dynamic property of the language, subject to change as the compiler evolves.

CHECKING FOR TYPE STABILITY

There are several mechanisms to determine whether a function call is type stable. One of them is based on the `@code_warntype` macro, which reports all the types inferred during a function call. To illustrate its use, consider a function that defines `y` as a transformation of `x`, and then uses `y` to perform some operation.

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

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

```
julia> @code_warntype foo(1.)
```

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

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

```
julia> @code_warntype foo(1)
```

The output of `@code_warntype` can be difficult to interpret. Nonetheless, the addition of colors facilitates its understanding:

- If all lines are **blue**, the function is **type stable**. This means that Julia can identify a unique concrete type for each variable.
- If at least one line is **red**, the function is **type unstable**. It reflects that one variable or more could potentially adopt multiple possible types.
- **Yellow** lines indicate type instabilities that the compiler can handle effectively, in the sense that they have a reduced impact on performance. As a rule of thumb, **you can safely ignore them**.

> **Warning!**

> Throughout the website, we'll refer to **type instabilities** as those indicated by a red warning exclusively. Yellow warnings will be mostly ignored.

In the provided example, the compiler attempts to infer concrete types. This is done by identifying two pieces of information, given `x`'s concrete type:

i) the type of `y`,

ii) the type of `y * i` where `i` has type `Int64`, implicitly defining the type of `[y * i for i in 1:100]`.

The example clearly demonstrates that **the same function can be type stable or unstable depending on the types of its inputs**: `foo` is type stable when `x` has type `Int64`, but type unstable when `x` is `Float64`.

Specifically, in the scenario where `x = 1`, the compiler infers for *i)* that `y` can be equal to either `0` or `x`. Since both `0` and `1` are `Int64`, the compiler identifies a unique type for `y`, given by `Int64`. Regarding *ii)*, `y * i` also yields an `Int64`, as both `i` and `y` have type `Int64`. This determines that `[y * i for i in 1:100]` has type `Vector{Int64}`. Consequently, `foo(1)` is type stable, enabling Julia to invoke a method specialized for integers.

As for `x = 1.0`, the information for *i)* is that `y` could be either `0` or `1.0`. As a result, the compiler can't infer a unique type for `y`, which could be either `Int64` or `Float64`. The `@code_warntype` macro reflects this, identifying `y` as having type `Union{Float64, Int64}`. This ambiguity affects *ii)*, forcing the compiler to consider approaches that handle both `Float64` and `Int64`, and hence preventing specialization. Overall, `foo(1.0)` is type unstable, which has a detrimental impact on performance.

> **Remark**
>
> The conclusions regarding type stability wouldn't have changed if we had considered, for instance, `foo(-2)` or `foo(-2.0)`. This is because the compilation process relies on information about types, not values. More specifically, this means that type stability depends on whether `x` has type `Int64` or `Float64`, regardless of its actual value.

YELLOW WARNINGS MAY TURN RED

Not all instances of type instabilities have the same impact on performance. Their severity is ultimately indicated through a yellow or red warning. Yellow warnings denote a relatively minor impact on performance, typically resulting from isolated computations that Julia can handle effectively. However, repeated execution of these operations may escalate into more serious performance issues, triggering a red warning. The following example demonstrates a scenario like this.

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

    y * 2
end
```

```
julia> @code_warntype foo(1.)
```

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

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

```
julia> @code_warntype foo(1.)
```

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

    for i in 1:100
      y = y + i
    end

    return y
end
```

```
julia> @code_warntype foo(1.)
```

The yellow warning reflects that `y * 2` could return either a `Float64` or `Int64` value. However, this operation is computed only once and based on two types that the compiler can handle efficiently. Instead, the second tab involves multiple computations `y * i` without knowledge of a unique concrete type for `y`, resulting in a red warning.

Despite this, note that not all yellow warnings will necessarily escalate to a red warning when incorporated into a for-loop. The third tab illustrates this point, reinforcing that not all type instabilities are equally detrimental.

> **For-Loops and Yellow Warnings**
>
> A yellow warning will always be displayed when running a for-loop, even if the operation itself is type stable. In such cases, the warning can safely be disregarded, as it simply reflects the inherent behavior of iterators: they return either the next element to iterate over or `nothing` with type `Nothing` when the sequence is exhausted.
>
> ▼ For-Loop

```
function foo()
    for i in 1:100
        i
    end
end
```

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