7d. Preliminaries on Types

[Martin Alfaro](#)

PhD in Economics

INTRODUCTION

High performance in Julia is intimately related to the notion of type stability. The definition of this concept is relatively straightforward: a function is type-stable when the types of its expressions can be inferred from the types of its arguments. When this property holds, Julia can specialize its computation method, resulting in fast code.

Despite its simplicity, type stability is subject to various nuances. In fact, a careful consideration of the property requires a solid foundation in two key areas: **Julia's type system and the inner workings of functions**. The current section equips you with the necessary knowledge to grasp the former, deferring the internals of functions to the next section. Moreover, **the explanations will focus on the case of scalars and vectors**, leaving more complex objects for subsequent sections.

Before you continue, I recommend reviewing the basics of types introduced [here](#).

> **Warning!**
>
> **The subject is covered only to the extent necessary for understanding type stability**. Julia's type system is indeed quite vast, and a comprehensive exploration would warrant a dedicated chapter.

BASICS OF TYPES

Variables in Julia serve as mere labels for objects, with objects in turn holding values of specific types. The most common types for scalars are `Float64` and `Int64`, whose vector counterparts are `Vector{Float64}` and `Vector{Int64}`. Recall that `Vector` is an alias for a one-dimensional array, so that a type like `Vector{Float64}` is equivalent to `Array{Float,1}`.

> `Int` **As an Alternative to** `Int64`
>
> You'll notice that packages tend to use `Int` as the default type for integers. The type `Int` is an alias that adapts to your CPU's architecture. Since most modern computers are 64-bit systems, `Int` is equivalent to `Int64`. Nonetheless, `Int` becomes `Int32` on 32-bit systems.

Julia's type system is organized in a hierarchical way. This feature permits the definition of subsets and supersets of types, which in the context of types are referred to as **subtypes** and **supertypes**. [1] For instance, the type `Any` is a supertype that includes all possible types in Julia, thus occupying the highest position in the type hierarchy. Another example of supertype is `Number`, which encompasses all numeric types (`Float64`, `Float32`, `Int64`, etc.).

Supertypes provide great flexibility for writing code. They enable the grouping of values under a common abstraction, making it possible to define operations generically. For instance, defining `+` for the abstract type `Number` ensures its applicability to all numeric types, regardless of whether they are integers, floats, or their numerical precision.

A particular supertype known as `Union` will be instrumental for our examples. It allows variables to hold values of any type specified in its arguments. Its syntax is `Union{<type1>, <type2>, ...}`, so that a variable with type `Union{Int64, Float64}` could be either an `Int64` or `Float64`. Note that, by definition, union types are always supertypes of their constituent types.
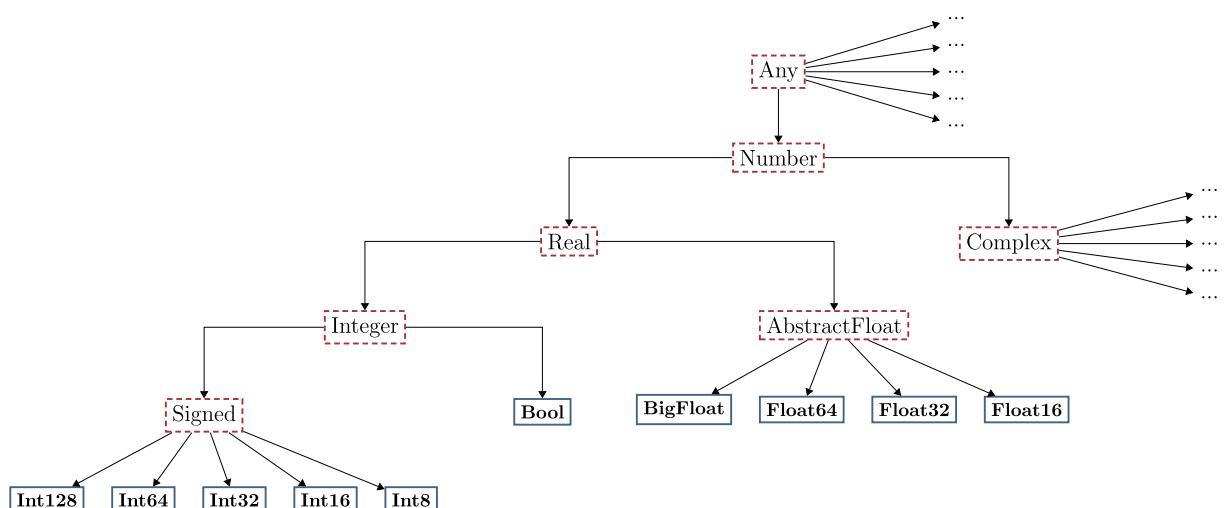
ABSTRACT AND CONCRETE TYPES

The hierarchical nature of types makes it possible to represent subtypes and supertypes as trees. Such structure gives rise to the notions of abstract and concrete types.

An **abstract type** acts as a parent category, necessarily breaking down into subtypes. The type `Any` in Julia is a prime example. In contrast, a **concrete type** represents an irreducible unit that therefore lacks subtypes. Concrete types are considered final, in the sense that they can't be further specialized within the hierarchy.

The diagram below illustrates the difference between abstract and concrete types for scalars. In particular, we present the hierarchy of the type `Number`, where the labels included match the corresponding type name in Julia. [2]
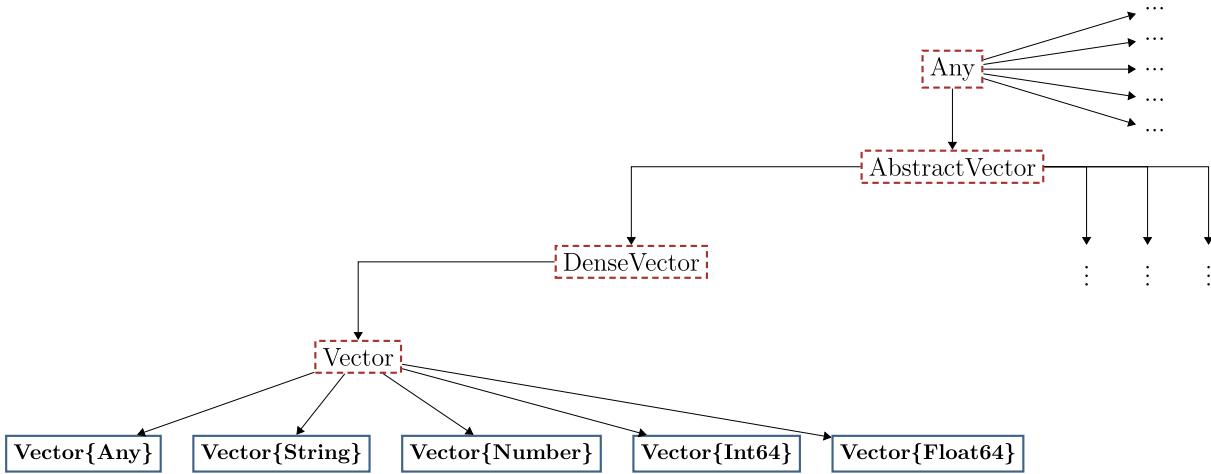
HIERARCHY OF TYPE NUMBER

Note: Dashed red borders indicate abstract types, while solid blue borders indicate concrete types.

The distinction between abstract and concrete types for scalars is relatively straightforward. Instead, the same distinction becomes more nuanced when vectors are considered, as shown in the diagram below.

HIERARCHY OF TYPE VECTOR

Note: Dashed red borders indicate abstract types, while solid blue borders indicate concrete types.

The tree reveals that `Vector{T}` **for a given type** `T` **is a concrete type**. This means that variables can be instances of `Vector{T}`, where `Vector{T}` doesn't have subtypes. The consequence is that a vector like `Vector{Int64}` isn't a subtype of `Vector{Any}`, even though `Int64` is a subtype of `Any`. This behavior stands in stark contrast to scalars, where `Any` is an abstract type. However, it aligns perfectly with the concept of vectors as collections of *homogeneous elements*, meaning that all elements share the same type.

ONLY CONCRETE TYPES CAN BE INSTANTIATED, ABSTRACT TYPES CAN'T

Instantiation refers to the process of creating an object with a specific type. A key principle of Julia's type system is that **only concrete types can be instantiated**, implying that values can never be represented by abstract types. This distinction helps clarify the meaning of some widespread expressions used in Julia. For example, stating that a variable has type `Any` shouldn't be interpreted literally. Rather, it means the variable can hold values of any concrete type, considering that all concrete types are subtypes of `Any`.

This distinction will become crucial for what follows, particularly for type-annotating variables. It implies that declaring a variable with an abstract type restricts the set of possible concrete types it can hold, even though the variable will ultimately adopt a concrete type.

RELEVANCE FOR TYPE STABILITY

At this point, you may be wondering how all these concepts relate to type stability. The connection becomes clear when you consider how Julia performs computations.

High performance in Julia relies heavily on specializing the computation method. We'll see in the next section that this specialization is unattainable in the global scope, as Julia treats global variables as potentially holding values of any type. In contrast, when code is wrapped in a function, the execution process begins by determining the concrete types of each function argument. This information is then used to infer the concrete types of all the expressions within the function body.

When this inference succeeds and all expressions have unambiguous concrete types, the function is considered **type stable**. This enables Julia to specialize its computation method and generate optimized machine code. If, instead, expressions could potentially take on multiple concrete types, performance is substantially degraded, as Julia must consider a separate implementation for each possible type.

For scalars and vectors, type stability essentially requires that expressions ultimately operate on **primitive types**. Examples of numeric primitive types include integers and floating-point numbers, such as `Int64`, `Float64`, and `Bool`. Thus, applying functions like `sum` to a `Vector{Int64}` or `Vector{Float64}` allows for full specialization, whereas applying them to a `Vector{Any}` prevents it.

> **String Objects**
>
> For text representation, the character type `Char` serves as a primitive type. Since a `String` is internally represented as a collection of `Char` elements, operations on `String` objects can also achieve type stability.

THE OPERATOR <: TO IDENTIFY SUPERTYPES

The remainder of this section is dedicated to operators and functions for handling types. Specifically, we'll introduce the operator `<:`, which checks whether one type is a subtype of another. Then, we'll examine strategies for constraining variables to specific types.

It's possible that you won't need to apply any of the techniques we present, as Julia automatically attempts to infer types when functions are called. Nonetheless, understanding these operators is essential for grasping upcoming material.

USE OF `<:`

The symbol `:<` tests whether a type `T` is a subtype of another type `S`. It can be used as an operator `T <: S` or as a function `<:(T,S)`. For example, `Int64 <: Number` and `<:(Int64, Number)` verify whether `Int64` is a subtype of `Number`, thus returning `true`. Below, we provide further examples.

```
# all the statements below are `true`
Float64 <: Any
Int64   <: Number
Int64   <: Int64
```

```
# all the statements below are `false`
Float64 <: Vector{Any}
Int64   <: Vector{Number}
Int64   <: Vector{Int64}
```

The fact that `Int64 <: Int64` evaluates to `true` illustrates a fundamental principle: **every type is a subtype of itself**. Moreover, in the case of concrete types, this is the only subtype.

THE KEYWORD `WHERE`

By combining `<:` with `Union`, you can also check whether a type belongs to a given set of types. For example, `Int64 <: Union{Int64, Float64}` assesses whether `Int64` equals `Int64` or `Float64`, thus returning `true`.

The approach can be made more widely applicable by using the `where` keyword with a type parameter `T`. [3] The syntax is `<type depending on T> where T <: <set of types>`. This entails that `T` covers multiple possible types.

```
# all the statements below are `true`
Float64 <: Any
Int64   <: Union{Int64, Float64}
Int64   <: Union{T, String} where T <: Number      # `String` represents text
```

```
# all the statements below are `true`
Vector{Float64} <: Vector{T} where T <: Any
Vector{Int64}   <: Vector{T} where T <: Union{Int64, Float64}
Vector{Number}  <: Vector{T} where T <: Any

# all the statements below are `false`
Vector{Float64} <: Vector{Any}
Vector{Int64}   <: Vector{Union{Int64, Float64}}
Vector{Number}  <: Vector{Any}
```

```
# all the statements below are `true`
Vector{Float64} <: Vector{<:Any}
Vector{Int64}   <: Vector{<:Union{Int64, Float64}}
Vector{Number}  <: Vector{<:Any}

# all the statements below are `false`
Vector{Float64} <: Vector{Any}
Vector{Int64}   <: Vector{Union{Int64, Float64}}
Vector{Number}  <: Vector{Any}
```

Types relying on parameters like `T` are called **parametric types**. In the example above, these types allow us to distinguish between a concrete type like `Vector{Any}` and a set of concrete types `Vector{T} where T <: Any`, where the latter encompasses `Vector{Int64}`, `Vector{Float64}`, `Vector{String}`, etc.

> **Warning!** - The Type `Any`
>
> When we omit `<:` and simply write `where T`, Julia implicitly interprets the statement as `where T <: Any`. This is why the following equivalences hold.
>
> ```
> # all the statements below are `true`
> Float64        <: Any
> Float64          <: T where T <: Any            # identical to
> the line above
> Vector{Int64} <: Vector{T} where T <: Any
> ```
>
> ```
> # all the statements below are `true`
> Float64        <: Any
> Float64          <: T where T                   # identical to
> the line above
> Vector{Int64} <: Vector{T} where T
> ```

TYPE-ANNOTATING VARIABLES

In the following, we present methods for **type-annotating variables**. The techniques introduced can be used either to assert a variable's type **during an assignment** or to restrict the types of **function arguments**.

There are two approaches to type-annotating variables. The first one relies on the binary operator `::`, and its syntax is `x::<type>`. The second approach leverages the Boolean binary operator `<:`, combined with `::` and the keyword `where`. Its syntax is `x::T where T <: <type>`, where `T` can be replaced with other symbol.

Next, we illustrate both methods, separately considering type-annotations for assignments and function arguments.

ASSIGNMENTS

Let's start illustrating the approaches based on scalar assignments. Each tab below declares an identical type for `x` and for `y`.

```julia
x::Int64              = 2        # only reassignments to `Int64` are possible

y::Number             = 2         # only reassignments to `Float64`, `Float32`, `Int64`, etc
are possible
```

```julia
julia> x = 2.5
ERROR: InexactError: Int64(2.5)

julia> y = 2.5
2.5

julia> y = "hello"
ERROR: MethodError: Cannot convert an object of type String to an object of type Number
```

```julia
x::T where T <: Int64  = 2        # only reassignments to `Int64` are possible

y::T where T <: Number = 2         # only reassignments to `Float64`, `Float32`, `Int64`, etc
are possible
```

```julia
julia> x = 2.5
ERROR: InexactError: Int64(2.5)

julia> y = 2.5
2.5

julia> y = "hello"
ERROR: MethodError: Cannot convert an object of type String to an object of type Number
```

> **Warning!** - Modifying Types
>
> Once a type for `x` has been assigned, the type can't be changed. The only way to fix this is by starting a new session.

The fact that `x` holds the same type across all tabs follows because `T <: Float64` can only represent `Float64`. More specifically, `Float64` is a concrete type, which by definition has no subtypes other than itself. Considering this, scalar types are usually asserted using `::` rather than `<:`.

While this behavior holds for scalars, it doesn't apply to vectors. Specifically, using `::` in combination with `Vector{Number}` establishes that the object will have `Vector{Number}` as its concrete type. Instead, `Vector{T} where T <: Number` indicates that the elements of the vector will adopt a concrete subtype of `Number`.

```
# `x` will always be `Vector{Any}`
x::Vector{Any}            = [1,2,3]

# `y` will always be `Vector{Number}`
y::Vector{Number}         = [1,2,3]
```

```
julia> typeof(x)
Vector{Any} (alias for Array{Any, 1})

julia> typeof(y)
Vector{Number} (alias for Array{Number, 1})
```

```
# `x` is Vector{Int64} and could eventually become `Vector{Float64}`, `Vector{String}`, etc
x::Vector{T} where T <:Any      = [1,2,3]

# `x` is Vector{Int64} and could eventually become `Vector{Float64}`, `Vector{Int32}`, etc
y::Vector{T} where T <: Number = [1,2,3]
```

```
julia> typeof(x)
Vector{Int64} (alias for Array{Int64, 1})

julia> typeof(y)
Vector{Int64} (alias for Array{Int64, 1})
```

The principles outlined apply even when a variable isn't explicitly type-annotated. The reason is that **an assignment without** `::` **implicitly assigns the type** `Any` **to the variable**. For example, the statements `x = 2` and `x::Any = 2` are equivalent.

The same occurs when omitting `<:` from the expression `where T`, which implicitly takes `T <: Any`. Thus, for instance, `x = 2` is equivalent to `x::T where T = 2` or `x::T where T <: Any = 2`. Considering this, all variables listed below have their types constrained in a similar manner.

```
# all are equivalent
a      = 2
b::Any = 2
```

```
# all are equivalent
a                  = 2
b::T where T       = 2
c::T where T <: Any = 2
```

Once we recognize that variables default to the type `Any`, it becomes clear why they can be reassigned with values of different types. For instance, given `a = 1`, executing `a = "hello"` afterwards is valid, since `a` is implicitly type-annotated with `Any`.

> **Warning!** - One-liner Statements Using `where`
>
> Be careful with one-liner statements using `where`, especially when `where T` is shorthand for `where T <: Any`. These concise statements can easily lead to confusion, as demonstrated below.

```
a::T where T = 2                      # this is not `T = 2`, it's
`a = 2`

a::T where {T}         = 2            # slightly less confusing
notation
a::T where {T <: Any} = 2            # slightly less confusing
notation
```

```
foo(x::T) where T = 2                # this is not `T = 2`, it's
`foo(x) = 2`

foo(x::T) where {T}         = 2      # slightly less confusing
notation
foo(x::T) where {T <: Any} = 2      # slightly less confusing
notation
```

FUNCTIONS

Function arguments can also be type-annotated. This is illustrated below, where functions are restricted to integer inputs exclusively.

```
function foo1(x::Int64, y::Int64)
    x + y
end
```
```
julia> foo1(1, 2)
3
julia> foo1(1.5, 2)
ERROR: MethodError: no method matching foo1(::Float64, ::Int64)
```

```
function foo2(x::Vector{T}, y::Vector{T}) where T <: Int64
    x .+ y
end
```
```
julia> foo2([1,2], [3,4])
2-element Vector{Int64}:
  4
  6
julia> foo2([1,2], [3.0, 4.0])
ERROR: MethodError: no method matching foo2(::Vector{Int64}, ::Vector{Float64})
```

Note that when both function arguments are annotated with the same type parameter `T`, they're constrained to share exactly the same type. Also notice that types like `Int64` preclude the use of `Float64`, even for numbers like `3.0`. Considering both facts, greater flexibility can be achieved by introducing separate type parameters, annotating them with a common abstract type like `Number`.

```julia
function foo2(x::T, y::T) where T <: Number
    x + y
end
```

```
julia> foo2(1.5, 2.0)
3.5

julia> foo2(1.5, 2)
ERROR: MethodError: no method matching foo2(::Float64, ::Int64)
```

```julia
function foo3(x::T, y::S) where {T <: Number, S <: Number}
    x + y
end
```

```
julia> foo3(1.5, 2.0)
3.5

julia> foo3(1.5, 2)
3.5
```

The greatest flexibility is achieved when we don't type-annotate function arguments at all, as they'll implicitly default to $\boxed{\text{Any}}$. This can be observed below, where all tabs define identical functions. Ultimately, type-annotating function arguments is only needed to prevent invalid usage (e.g., to ensure that $\boxed{\text{log}}$ isn't applied to a negative value).

```julia
function foo(x, y)
    x + y
end
```

```julia
function foo(x::Any, y::Any)
    x + y
end
```

```julia
function foo(x::T, y::S) where {T <: Any, S <: Any}
    x + y
end
```

```julia
function foo(x::T, y::S) where {T, S}
    x + y
end
```

DEFINING VARIABLES WITH CERTAIN TYPE

To conclude this section, we present an approach for converting values into a specific type. The approach makes use of the so-called **constructors**, which are functions that create new instances of a concrete type. They're useful for transforming a variable $\boxed{\text{x}}$ into another type.

Constructors are implemented via functions of the form `Type(x)`, where `Type` should be replaced with the name of the type (e.g., `Vector{Float64}`). Like any other function, constructors also support broadcasting.

```
x = 1

y = Float64(x)
z = Bool(x)
```
```
julia> y
1.0
julia> z
true
```

```
x = [1, 2, 3]

y = Vector{Any}(x)
```
```
julia> y
3-element Vector{Any}:
 1
 2
 3
```

```
x = [1, 2, 3]

y = Float64.(x)
```
```
julia> y
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

> **Remark**
>
> Parametric types can be used as constructors. Moreover, abstract types may still serve as constructors, despite that they can't be instantiated. In such cases, Julia will attempt to convert the object to a specific concrete type. Nonetheless, not all abstract types can be used for this purpose.
>
> ```
> x = 1
>
> y = Number(x)
> ```
> ```
> julia> typeof(y)
> Int64
> ```

```
x = [1, 2]

y = (Vector{T} where T)(x)
```

```
julia> typeof(y)
Vector{Int64}
```

```
x = 1

z = Any(x)
```

```
ERROR: MethodError: no constructors have been defined
for Any
```

An alternative to transform x 's type into T is given by the function convert(T,x) . Note that this method only works when a valid conversion exists, such as when Float64 can be translated into an equivalent Int64 (e.g., 3.0).

```
x = 1

y = convert(Float64, x)
z = convert(Bool, x)
```

```
julia> y
1.0
julia> z
true
```

```
x = [1, 2, 3]

y = convert(Vector{Any}, x)
```

```
julia> y
3-element Vector{Any}:
 1
 2
 3
```

```
x = [1, 2, 3]

y = convert.(Float64, x)
```

```
julia> y
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

FOOTNOTES

[1.] Types don't necessarily follow a subtype-supertype hierarchy. For example, `Float64` and `Vector{String}` exist independently, without a hierarchical relationship. This fact will become clearer when the concepts of abstract and concrete types are defined.

[2.] The `Signed` subtype of `Integers` allows for the representation of negative and positive integers. Julia also offers the type `Unsigned`, which only accepts positive integers and comprises subtypes such as `UInt64` and `UInt32`.

[3.] `T` can be replaced by any other letter