7d. Preliminaries on Types

Martin Alfaro
PhD in Economics

INTRODUCTION

High performance in Julia depends critically on 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 the property holds, Julia can specialize its computation method, resulting in significant performance gains.

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. **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, where objects in turn hold values with certain 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 allows for 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 any 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 to define operations in common. 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 special supertype known as `Union` will be instrumental for our examples. This construction is useful for variables that can potentially hold values with different types. They're denoted by `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 arguments.

> **Union of Types to Account for Missing Values**
>
> Unions of types emerge naturally in data analysis workflows, especially when handling missing observations. In Julia, these values are represented by the type `Missing`. For instance, if we load a column that contains both integers and empty entries, this is usually stored with type `Vector{Union{Missing,Int64}}`.

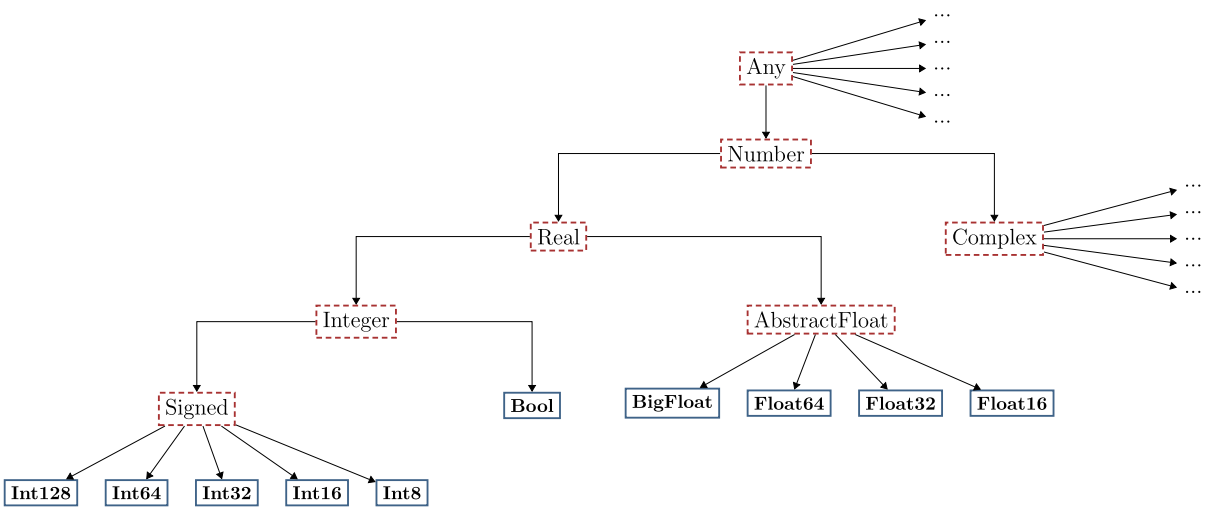
ABSTRACT AND CONCRETE TYPES

The hierarchical nature of types makes it possible to represent subtypes and supertypes as trees. Th 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` i Julia is a prime example. In contrast, a **concrete type** represents an irreducible unit that therefor lacks subtypes. Concrete types are considered final, in the sense that they can't be further specialize within the hierarchy.

The diagram below illustrates the difference between abstract and concrete types for scalars. This done by presenting the hierarchy of the type `Number`, where the labels included match th corresponding type name in Julia. [2]

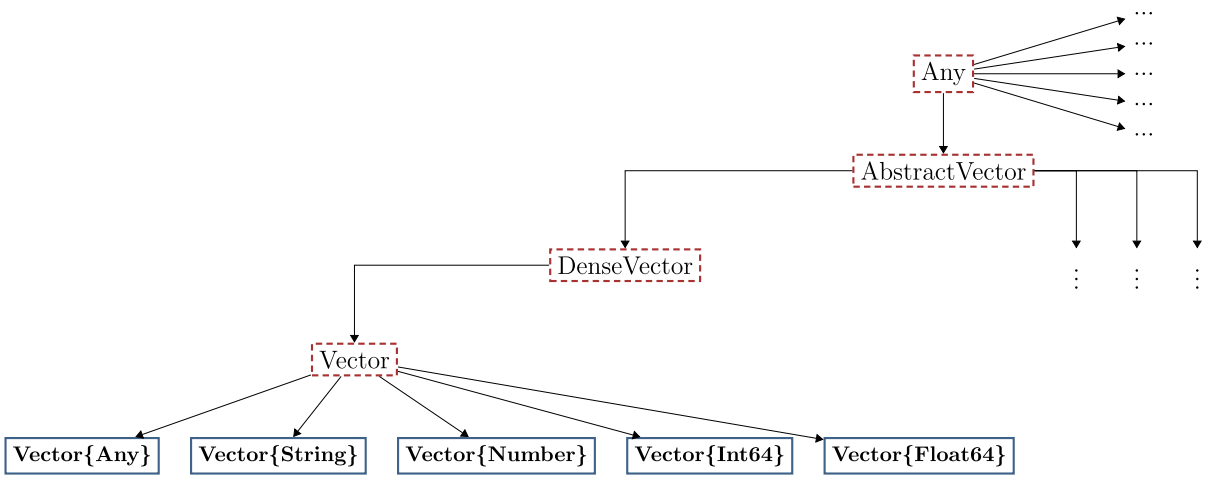
HIERARCHY OF TYPE NUMBER



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**. By definition, this mean variables can be instances of `Vector{T}` and `Vector{T}` can't have subtypes. The latter in particula implies that a vector like `Vector{Int64}` isn't a subtype of `Vector{Any}`, even though `Int64` is subtype of `Any`. This behavior stands in stark contrast to scalars, where `Any` is an abstract typ However, it aligns perfectly with the concept of vectors as collections of *homogeneous element* meaning they all share the same type.

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

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

This distinction will become crucial for what follows, particularly for type-annotating variables. implies that declaring a variable with an abstract type restricts the set of possible concrete types it ca 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 connectio becomes clear when you consider how Julia performs computations.

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

When this inference succeeds, meaning all expressions have unambiguous concrete types, th function is considered **type stable**. TType stability enables Julia to specialize its computation metho and generate optimized machine code. If, instead, expressions could potentially take on multipl concrete types, performance is substantially degraded, as Julia must consider a separat implementation for each possible type.

For scalars and vectors, type stability essentially requires that expressions ultimately operate o **primitive types**. Examples of numeric primitive types include integers and floating-point number such as `Int64`, `Float64`, and `Bool`. Thus, applying functions like `sum` to a `Vector{Int64}` o `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 the 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 rest of this section is dedicated to operators and functions for working with types. Specifically, we'll introduce the operator `<:`, which checks whether a given type is a subtype of another, and then explore ways to constrain a variable to certain 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`, which would return `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>`. In this way, `T` represents multiple possibilities.

```
# 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 that add parameters like `T` are called **parametric types**. In the example above, they 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 we can establish the following equivalences.

```
# 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**.

Specifically, 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>` (note that `T` accepts any other letter).

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

ASSIGNMENTS

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

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

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

```
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
```

```
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> 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 you assert a type for `x` in an assignment, you can't modify `x`'s type afterwards. The only way to fix this is by starting a new Julia session.

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

On the contrary, the implications when `::` or `<:` is chosen differs for vectors. Specifically, using `::` in combination with `Vector{Number}` establishes that `Vector{Number}` is the only possible concrete type. Instead, `Vector{T} where T <: Number` indicates that the elements of the vector will adopt a concrete type that's a subtype of `Number`, rather than the object adopting `Vector{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**, where `Any` is the supertype encompassing all possible types. 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 the variables below have their types restricted in the same way.

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

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

The default restriction of variables to the type `Any` is the reason why we can reassign variables with any value. 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. The examples below illustrate this by restricting the function to accept 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 type-annotating both arguments with the same parameter `T` forces them to have exactly the same type. Also notice that types like `Int64` preclude the use of `Float64`, even for numbers like `3.0`. If you need greater flexibility, you should introduce different type parameters and annotate them with an abstract type like `Number`.

```
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)
```

```
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 will implicitly default to `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 `log` isn't applied to a negative value).

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

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

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

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

CREATING VARIABLES WITH SOME TYPE

To conclude this section, we present an approach to defining variables with a given type. The approach relies on the so-called **constructors**, which are functions that create new instances of a concrete type. They're useful for transforming a variable `x` into another type.

Constructors are implemented by functions of the form `Type(x)`, where `Type` should be replaced with the literal name of the type (e.g., `Vector{Float64}`). Just like any other function, `Type` supports 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, although abstract types can't be instantiated, they may still serve as constructors. In such cases, Julia will attempt to convert the object to a specific concrete type, although 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
```

There's an alternative way to transform `x`'s type into `T`, as long as the conversion is feasible. This is given by the function `convert(T,x)`.

```
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