7e. Functions: Type Inference and Multiple Dispatch

Martin Alfaro
PhD in Economics

INTRODUCTION

In Julia, functions are key for achieving high performance. This is by design, as functions have been engineered from the outset to generate efficient machine code.

However, to fully unlock their potential, we must first understand the underlying process of function calls. Essentially, when a function is called, Julia attempts to identify concrete types for its variables, eventually selecting the most suitable method for the function. At the heart of the process are three interconnected mechanisms: **dispatch**, **compilation**, and **type inference**. This section will provide a detailed explanation of each concept.

FUNCTION'S VARIABLE SCOPE

To fully appreciate why functions in Julia are central to performance, we must first revisit the notion of variable scope. A function's **local variables** encompass all variables defined within a function's scope, including its arguments and any variable created inside the function body. These variables exist only during the function's execution and are inaccessible from outside. **Global variables**, on the other hand, refer to any variable defined outside a function's scope and remain accessible throughout the program's execution.

From a practical perspective, a key takeaway from this section is that **wrapping code in functions is crucial for high performance in Julia**. Instead, relying on global variables imposes a significant performance penalty.

Note that the use of global variables isn't confined to operations in the global scope. It also takes place when a function references variables that aren't passed as arguments. This is why the detrimental effect of global variables will appear in all the following cases.

GLOBAL SCOPE

```
x     = 2

y     = 3 * x
```

```
julia> y
6
```

FUNCTION USING A GLOBAL VARIABLE

```
x      = 2

foo() = 3 * x
```

```
julia> foo()
6
```

Recall that an expression like `x = 2` is shorthand for `x::Any = 2`, reflecting that global variables default to `Any` if they aren't explicitly type-annotated. Also remember that only concrete types can be instantiated, meaning that values can only adopt a concrete type. Consequently, `x::Any` shouldn't be understood as `x` having type `Any`, but rather that `x` can take on any concrete type that's a subtype of `Any`. Since `Any` sits at the top of Julia's type hierarchy, this simply means that `x`'s types are unrestricted.

Working with a global variable like `x` prevents specialization of `*`. The reason is that Julia treats global variables as potentially embodying any value and therefore any type. The underlying logic for this behavior is that, even if a variable holds some value at a specific moment, the user may reassign it at any point in the program. Because of this possibility, Julia must consider multiple possible methods for its computation, one for each possible concrete type of `x`. In practice, this results in Julia generating code with multiple branches, potentially involving type checks, conversions, and object creations. The consequence is degraded performance.

Even if we had type-annotated `x` with a concrete type like `x::Int64 = 2`, the performance limitations wouldn't completely go away. Certain optimizations can only be implemented when both the scope of variables is clearly delimited and values are known. When this is the case, Julia can gain a comprehensive view of all the operations to be performed, creating opportunities for further optimizations.

Functions in Julia were designed to address all these considerations. They achieve this following a series of steps, which we cover next.

FUNCTIONS AND METHODS

A **function** is a name that can be associated with multiple implementations known as methods. A **method** defines a specific function body for a given combination of argument types and number of arguments. The list of methods associated with a function `foo` can be retrieved by running the command `methods(foo)`.

To make this concrete, let's define several methods for some function `foo`. Creating methods requires type-annotating `foo`'s arguments with the operator `::` during its definition. By using this strategy, we can set a distinct body function for each unique combination of types.

To keep matters simple, let's start considering a scenario where all the methods of `foo` have the same number of arguments.

METHODS

```
foo(a,b)                    = a + b
foo(a::String, b::String) = "This is $a and this is $b"
```

```
julia> methods(foo)
2 methods for generic function "foo" from Main
julia> foo(1,2)
3
julia> foo("some text", "more text")
"This is some text and this is more text"
```

Since `foo(a,b)` is equivalent to `foo(a::Any,b::Any)`, the first method sets the behavior of `foo` for any combination of input types. Such behavior is subsequently overridden by the creation of the method `foo(a::String, b::String)`. By doing this, we provide an alternative function body for `a` and `b` with type `String`. The existence of multiple methods explains the differences in outputs obtained: the first method of `foo` is called with `foo(1, 2)`, whereas `foo("some text", "more text")` triggers the second method.

The example also reveals that **methods don't need to comprise similar operations**. Although mixing drastically different operations under a single function name isn't recommended, allowing function bodies to differ is relevant for performance. In particular, it's commonly used for tailoring the computation algorithms to each specific type combination, thus optimizing the overall performance of a function.

Also note that **methods don't need to have the same number of arguments**. For instance, it's possible to define the following methods for a function `bar`.

METHODS WITH DIFFERENT NUMBERS OF ARGUMENTS

```
bar(x)       = x
bar(x, y)    = x + y
bar(x, y, z) = x + y + z
```

```
julia> methods(bar)
3 methods for generic function "bar" from Main
julia> bar(1)
1
julia> bar(1, 2)
3
julia> bar(1, 2, 3)
6
```

Defining methods with different number of arguments is particularly useful for extending a function's behavior. A prime application is given by the function `sum`. So far, we've only used its simplest form `sum(x)`, which adds all the elements of a collection `x`. However, `sum` also supports additional methods. One of them is `sum(<function>, x)`, where the elements of `x` are transformed via `<function>` before being summed.

METHODS FOR 'SUM'

```
x = [2, 3, 4]

y = sum(x)          # 2 + 3 + 4
z = sum(log, x)     # log(2) + log(3) + log(4)
```

FUNCTION CALLS

Building upon our understanding of function definition and methods, let's now analyze the process triggered when a function is called. In the following, all our explanations will be based on the following function `foo`:

EXAMPLE FUNCTION 'FOO'

```
foo(a, b) = 2 + a * b
```

```
julia> foo(1, 2)
4
julia> foo(3, 2)
8
julia> foo(3.0, 2)
8.0
```

In Julia, defining a function like `foo(a, b)` is shorthand for creating a **method** with the signature `foo(a::Any, b::Any)`. Thus, the function body `foo(a,b)` holds for all possible type combinations of `a` and `b`.

When `foo(1, 2)` is called, Julia evaluates the expression `2 + a * b` by following a series of steps.

The process begins with what's known as **multiple dispatch**, where Julia selects which method of a function to execute. Importantly, this decision is based solely on the types of the arguments, not their values. Specifically, Julia begins by identifying the concrete types of the function arguments. In our example where `a = 1` and `b = 2`, both are identified as `Int64`. The information on types is then used to select a *method*, which defines the function body and hence the operations to be performed. This process involves searching through the available methods of `foo` to find the most specific one for the concrete types of `a` and `b`. In our example, `foo` has only one method `foo(a,b) = 2 + a * b`, which is defined for all argument types, including `a::Int64` and `b::Int64`. Therefore, the corresponding function body is `2 + a * b`.

The specific version of this method for the signature `foo(a::Int64, b::Int64)` is known as a **method instance**. If the code for the method instance `foo(a::Int64, b::Int64)` already exists, Julia will directly employ it to compute `foo(1,2)`. Otherwise, the compiler generates optimized code for that method instance, stores it in memory for future use, and Julia executes it.

The following diagram depicts the process unfolded when `foo(1,2)` is executed.

MULTIPLE DISPATCH

The process outlined has implications for how the language works. When a function is called with a particular combination of argument types for the first time, Julia incurs an additional cost because it must generate specialized code for those types. This initial delay is often referred to as Time To First Plot, a phrase that highlights how the compilation overhead becomes noticeable in interactive workflows such as plotting. Once the code has been compiled, however, Julia stores the resulting method instance, so that subsequent calls with the same argument types can reuse it. This eliminates the compilation step and leads to much faster execution.

The example with `foo` illustrates this process clearly. After evaluating `foo(1, 2)`, Julia has already compiled a method instance for the signature `foo(a::Int64, b::Int64)`. This is why the subsequent call `foo(3, 2)` can be executed immediately by invoking the cached method instance, without any need for recompilation. Instead, the execution of `foo(3.0, 2)` introduces a new combination of argument types, where `a::Float64` and `b::Int64`. Because no compiled method instance yet exists for this signature, Julia must generate one before executing the function.

TYPE INFERENCE

Most considerations for achieving high performance are related to the compilation process. In particular, Julia employs **Just-In-Time Compilation (JIT)**, a term reflecting that compilation happens on the fly during the function call.

A key mechanism in this process is **type inference**, whereby the compiler attempts to identify concrete types for *all* variables and expressions. If the compiler succeeds in identifying concrete types, it can specialize instructions for each operation and yield fast code.

This is the essence behind **type stability**, which we'll cover extensively in the next chapter. For instance, type inference in our example involves determining concrete types for `2`, `a = 1`, and `b = 2`. Since all values have type `Int64`, the compiler can specialize the computation of `2 + a * b` for variables with type `Int64`.

On the contrary, if the compiler is unable to identify concrete types for some expressions, it must create generic code to accommodate multiple combinations of types. This forces Julia to perform type checks and conversions during runtime, significantly degrading performance.

REMARKS ON TYPE INFERENCE

Below, we provide various remarks about type inference that are worth keeping in mind for next sections.

FUNCTIONS DO NOT GUARANTEE IDENTIFICATION OF CONCRETE TYPES

Merely wrapping code in a function doesn't guarantee that the compiler will identify concrete types for all operations. The following example illustrates this.

TYPE-UNSTABLE FUNCTION

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

foo(x) = x[1] + x[2]        # type unstable
```

```
julia> foo(x)
3
```

The issue in the example arises because the compiler assigns the type `Any` to both `x[1]` and `x[2]`, as they come from an object with type `Vector{Any}`. As a consequece, the compiler can't specialize the computation of the operation `+`. The example also highlights that compilation is exclusively based on types, not values. Thus, the generated code ignores the actual values `x[1] = 1` and `x[2] = 2`, which would otherwise indicate a type `Int64`.

GLOBAL VARIABLES INHERIT THEIR GLOBAL TYPE

Type inference is restricted to local variables, with any global variable having its type inherited from the global scope. For instance, consider the following example.

UNANNOTATED GLOBAL VARIABLE

```
a      = 2
b      = 1

foo(a) = a * b
```

```
julia> foo(a)
2
```

TYPE-ANNOTATED GLOBAL VARIABLE

```
a      = 2
b::Number = 1

foo(a) = a * b
```

```
julia> foo(a)
2
```

In both examples `b` is a global variable. Consequently, the compiler infers `b`'s type as `Any` in the first tab and as `Number` in the second tab.

TYPE-ANNOTATING FUNCTION ARGUMENTS DOES NOT IMPROVE PERFORMANCE

Identifying concrete types is crucial for achieving high performance. At first glance, this might suggest that explicitly annotating function arguments is necessary for performance, or at least beneficial. However, thanks to type inference, such annotations are redundant. In fact, adding them can be

counterproductive, as they unnecessarily restrict the types accepted by a function, thereby limiting its flexibility and potential applications.

To better appreciate this loss of flexibility, compare the following scripts.

UNANNOTATED FUNCTION

```
foo2(a, b) = a * b
```

```
julia> foo2(0.5, 2.0)
1.0
julia> foo2(1, 2)
2
```

TYPE-ANNOTATED FUNCTION

```
foo1(a::Float64, b::Float64) = a * b
```

```
julia> foo1(0.5, 2.0)
1.0
julia> foo1(1, 2)
ERROR: MethodError: no method matching foo1(::Int64, ::Int64)
```

The function on the first tab only accepts arguments of type `Float64`, implying that even integer variables are disallowed. By contrast, the function on the second tab mirrors the behavior with `Float64` inputs, but additionally allows for other types. This flexibility stems from the implicit type annotation `Any` on the function arguments.

> **Packages Commonly Type-Annotate Function Arguments**
>
> When inspecting the code of packages, you may notice that function arguments are often type-annotated. The reason for this isn't related to performance, but rather to ensure the function's intended usage, safeguarding against inadvertent type mismatches.
>
> For instance, suppose a function that computes the revenue of a theater via `nr_tickets * price`. Importantly, the operator `*` in Julia not only implements product of numbers, but also concatenates words when applied to expressions with type `String`. Without type-annotations, the function could potentially be misused. This is exemplified in the first tab below, with the second tab precluding this possibility by asserting types.
>
> **UNANNOTATED FUNCTION**
>
> ```
> revenue1(nr_tickets, price) = nr_tickets * price
> ```
>
> ```
> julia> revenue1(3, 2)
> 6
> julia> revenue1("this is ", "allowed")
> "this is allowed"
> ```

TYPE-ANNOTATED FUNCTION

```
revenue2(nr_tickets::Int64, price::Number) = nr_tickets * price
```

```
julia> revenue2(3, 2)
6
julia> revenue2("this is ", "not allowed")
ERROR: MethodError: no method matching revenue2(::String,
::String)
```