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, then 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.

FUNCTIONS AND GLOBAL VARIABLES

To fully grasp how functions enhance performance in Julia, it's essential to examine their relationship with variable scope. **Local variables** encompass all variables defined within a function's scope, including function arguments and variables declared inside the function body. These variables exist only during the function's execution and are inaccessible from outside the function. **Global variables**, on the other hand, refer to any variable defined outside a function's scope and and remain accessible throughout the program's execution.

One of the main takeaway of this section is that **wrapping operations within functions is crucial for achieving high performance in Julia**. Instead, working in the global scope, or more generally relying on global variables, precludes any performance implementation.

The low performance of global variables arises because Julia treats them as potentially embodying any value and therefore any type. This decision was adopted under the logic that, even if a variable holds some value at a specific moment, the user may reassign it at any part of the program.

The detrimental effect of global variables isn't confined to operations in the global scope. It also arises when a function references external variables that haven't been passed as arguments. Considering this, our conclusions apply to all the following cases.

GLOBAL VARIABLE IN GLOBAL SCOPE

```
x     = 2

y     = 3 * x
```

```
julia> y
6
```

GLOBAL VARIABLE IN FUNCTION

```
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` unless they're 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 is a subtype of `Any`. Since `Any` is at the top of Julia's type hierarchy, this simply means that `x`'s types are unrestricted.

Working with a variable like `x` that has type `Any`, prevents specialization of `*`. The reason is that 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. This is because certain optimizations can only be implemented when both the scope of variables is clearly delimited and their values are known. When both aspects are known, Julia can gain a comprehensive view of all the operations to be performed, creating opportunities for further optimizations.

Functions were designed in Julia to address all these considerations. This is accomplished through a series of steps that functions follow, which we cover next.

FUNCTIONS AND METHODS

A **function** is just a name that groups an arbitrary number of methods. Each **method**, in turn, defines a specific function body for a given combination of number and types of arguments. The list of methods associated with a function `foo` can be retrieved by running the command `methods(foo)`.

To illustrate these concepts, let's define several methods for some function named `foo`. To keep matters simple, let's start considering a scenario where all the methods in `foo` have the same number of arguments. Creating methods requires type-annotating `foo`'s arguments with the

operator `::` during its definition. Then, we can provide a distinct body function for each unique combination of these types.

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. However, such behavior is overridden by the method `foo(a::String, b::String)`, which provides an alternative function body for `a` and `b` with type `String`. The existence of multiple methods explains the different 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 by method creates opportunities for optimizations. In particular, it allows for computation algorithms tailored to each specific type combination, thus optimizing the overall performance of a function.

Additionally, note that **methods don't need to have the same number of arguments**. For instance, it's possible to define all 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
```

This feature 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 first 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 CALL

Given a function and its methods, we can now analyze the process triggered when a function is called. In the following, we'll base our explanations 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
```

Recall that variables with no type annotation default to `Any`. This implies that the function body `foo(a,b)` holds for any combination of types of `a` and `b`.

MULTIPLE DISPATCH

When `foo(1, 2)` is called, Julia is instructed to evaluate the expression `2 + a * b`. This process relies on a mechanism known as **multiple dispatch**, where Julia decides on the computational approach to be implemented. Importantly, this decision is based on solely on the types of the arguments, not their values.

Multiple dispatch proceeds in several steps. First, the compiler determines the concrete types of the function arguments. In our example, since `a = 1`, and `b = 2`, both are identified as `Int64`.

After this, the information on types is used to select a *method*, which defines the function body and hence the operations to be performed. This process involves searching through all available methods of `foo` until a method that matches the concrete types of `a` and `b` is identified. In our example, `foo` has only one method `foo(a,b) = 2 + a * b`, which applies to all type combinations of `a` and `b`. Consequently, the relevant function body is `2 + a * b`.

The operations to be performed are then forwarded to the **compiler**, which is in charge of the implementation. This involves choosing a **method instance**, which refers to the specific code implementation that will be used to compute the operations defined by the method.

If a method instance already exists for the function signature `foo(a::Int64, b::Int64)`, Julia will directly employ it to compute `foo(1,2)`. Otherwise, a new method instance is generated and stored (cached) in memory.

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

MULTIPLE DISPATCH

The process determines that the first time you call a function with particular argument types, there's an initial compilation overhead. This phenomenon is referred to in Julia as Time To First Plot. Instead, subsequent calls with the same argument types can reuse this cached code, resulting in faster execution.

To illustrate this mechanism, note that the call `foo(3, 2)` incorporated in the example occurs after `foo(1,2)`. This allows Julia to compute `foo(3, 2)` by directly invoking the method instance `foo(a::Int64, b::Int64)`, without the need of compiling code. Instead, executing a function call like `foo(3.0, 2)` requires the compilation of a new method instance `foo(a::Float64, b::Int64)`, since the types of `3` and `3.0` differ.

TYPE INFERENCE

During a function call, two different stages take place: compilation time and runtime. **Compilation time** consists of the steps just described, during which Julia generates machine code to execute the function's operations. Importantly, this stage involves no computations, and is triggered only when the function is called for the first time with new concrete types.

In contrast, **runtime** is the stage during which code instructions are actually executed. It takes place *after* compilation and every time a function is called.

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.

The quality of code generated during JIT critically determines performance. 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 of **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 of them 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, the compiler must generate generic code that accommodates multiple type possibilities. This forces Julia to defer decisions on methods to runtime, significantly degrading performance.

REMARKS ON TYPE INFERENCE

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

FUNCTIONS DO NOT GUARANTEE CONCRETE TYPES

Notice that merely wrapping operations in a function doesn't guarantee that the compiler will identify concrete types. The following example presents a function call that's unable to do so.

TYPE-UNSTABLE FUNCTION

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

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

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

In the example provided, the issue arises because the compiler assigns the type `Any` to `x[1]` and `x[2]`, since they correspond to elements from an object with type `Vector{Any}`. Consequently, 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 code is generated ignoring that actually `x[1] = 1` and `x[2] = 2`, and so are `Int64`.

GLOBAL VARIABLES INHERIT THEIR GLOBAL TYPE

Julia's attempt to identify concrete types is restricted to local variables. Instead, any global variable will have its type inherited from the global scope. For instance, consider the following example.

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, `b`'s type in the first tab is inferred to `Any`, while in the second tab to `Number`.

TYPE-ANNOTATING FUNCTION ARGUMENTS DOES NOT IMPROVE PERFORMANCE

Identifying concrete types is key for achieving performance. This might lead you to believe that type-annotating function arguments is essential for performance, or that at least could provide a boost. However, type-annotating arguments is actually redundant due to type inference. In fact, engaging in this practice will unnecessarily constrain the types accepted by the function, reducing the range of potential applications. To better appreciate this loss of flexibility, compare the following scripts.

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)
```

UNANNOTATED FUNCTION

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

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

The function on the first tab only accepts arguments with type `Float64`. Note that even integer variables are disallowed, as function arguments aren't converted to a common type. On the contrary, the function on the second tab entails the same process for `Float64` inputs, but additionally allows for other types. The flexibility stems from the implicit type-annotation `Any` for the function arguments.

> **Packages Commonly Type-Annotate Function Arguments**
>
> When inspect 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 to the revenue of a theater via `nr_tickets * price`. Importantly, the operator `*` in Julia not only implements the product of numbers, but also concatenates words when applied to expressions with type `String`. This opens up the possibility of misusing the function if it's not type-annotated. The first tab demonstrates a potential misuse of this function, with the second tab addressing 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)
```