

Multiple Dispatch

As we have seen in Julia, based on different input types we can define the same function with different methods. For example, let's make a function that computes $2x + y$ on **Int** and x/y on **Float64**:

```
# code chunk 20
foo(x::Int,y::Int) = 2x + y
foo(x::Float64,y::Float64) = x/y
@show foo(2,5) #9
@show foo(2.0,5.0) #0.4
```

Julia's multiple dispatch is really one of its defining features. A lot of people would tell you that Julia is not object oriented because it is multiple dispatch oriented, so there is dispatch style that is kind of in the background and this is why things are becoming fast. The reason why this is really the case is because the way that it's built up as a fast language on top of a JIT compiler is all based around the way that it changes the functions that it lowers to dependent on types. So multiple dispatch is actually just a very natural relation where as a user you're now allowed to tell Julia how it should be generating new functions based off the types. This is actually done all the way down at its very base. So when Julia opens up, first things it actually defines are integers and floating point numbers and those are defined within the language itself. Then what it does is defining **+(x::Float64,y::Float64)** and says this is an **llvm** call. so it actually just writes in the llvm at the lowest level and you can now know what operation it should be doing. But the nice thing this is actually defined in Julia programming language itself. It's not built into the compiler so even floating point arithmetic. So all of all those specializations all the way down are things that are written in the Julia language by using multiple dispatch. This is just it at its highest level as a feature but really what you can understand then is multiple dispatch is not really just a feature, rather this is actually the core way that the compiler is actually building optimized code for the inputs because it is now able to use this to essentially to deduce what the C code would produce if the input types are given.

```
# code chunk 20.1
@code_llvm foo(2,5)
```

So what is happening is you are writing C code except now it's able to know how to get this specialized versions. Like it knows **foo** on integers needs to call the specialized version for plus between an integer and an integer and it's going to call the specialized version of multiply for an integer and an integer and so on. The **+** function in Julia is just defined as **+(a,b)**, and we can actually point to that code in the Julia distribution:

```
# code chunk 20.2
@which +(2.0,5)

#+(x::Number, y::Number) in Base at promotion.jl:311

@which promote(2.0,5)
#promote(x, y) in Base at promotion.jl:280
```

and so on all the way down you can check everything is written Julia. So there is nothing hidden, everything that is done to make it faster are shown. Now for example you want to extend **foo(x::Int,y::Int) = 2x + y** to **Int32** we can add the line

```
# code chunk 20.3
foo(x::Union{Int,Int32},y::Int) = 2x + y
```

If you want to make your algorithms fast, then probably you have a different algorithm for float32, a different algorithm for float64 etc, but you have a compiler that kind of generate those different variants right. For example if you now write **2x + y** is going to be optimal for not just integers but also for all types, then someone already wrote the optimal way for doing plus or multiply, so you don't need to worry at this level like how do I make sure that it calls the optimal version of all those functions. Underneath the hood that is just doing multiple dispatch recursively. You get something fully specialized at the top level without worrying too much whats going under the hood.

Now let's look at the methods we already have for **foo**

```
# code chunk 21
methods(foo)
```

This will raise an error,

```
# code chunk 21.1
foo(2.0,5)
```

but if we write,

```
# code chunk 21.2
foo(x,y) = 2x+2y #same as foo(x::Any,y::Any) = 2x+2y
```

then its fine, but careful do you want it?

Ambiguities

The version that is called is actually the most strict version that is correct. What happens if it's impossible to define "the most strict version"? For example,

*(!!Kill the current terminal because we already defined a lot of **foo** to keep track)*

```
# code chunk 24
foo(x::Float64,y::Number) = 5x + 2y
foo(x::Number,y::Int) = x - y
```

What should it call on **f(2.0,5)** now? **foo(x::Float64,y::Number)** and **foo(x::Number,y::Int)** are both more strict than **foo(x::Number,y::Number)**. So one of them should be called, but neither are more strict than each other, and thus you will end up with an **ambiguity error**:

```
# code chunk 25
foo(2.0,5)
```

One way to fix this is to define a method lower than both of the above. Look at the error description, it already suggest that **Possible fix, define foo(::Float64, ::Int64)** So you define something like

```
# code chunk 25.1
foo(x::Float64, y::Int) = 2x - y
foo(2.0,5)
```

and then its fine

Function Barriers (a hack!)

Since functions automatically specialize on their input types in Julia, we can use this to our advantage in order to make an inner loop fully inferred. For example,

```
x = Number[2.0, 4.0] ## this needs some explanation
f(x, y) = x + y
function r(x)
    a = 4
    b = 2
    for i in 1:100
        c = f(x[1], a)
        d = f(b, c)
        a = f(d, x[2])
    end
    a
end
```

```
end
@btime r(x)
```

In here, the loop variables are not inferred and thus this is really slow. However, we can force a function call in the middle to end up with specialization and in the inner loop be stable. This might look very strange at first

```
s(x) = _s(x[1], x[2])

function _s(x1, x2)
    a = 4
    b = 2
    for i in 1:1000
        c = f(x1, a)
        d = f(b, c)
        a = f(d, x2)
    end
    a
end
@btime s(x)
```

Huge difference, so what is going on? Notice that this algorithm still doesn't infer the outer part.

```
@code_warntype s(x)
```

since the output of `_s` isn't inferred, but while it's in `_s` it will have specialized on the fact that `x[1]` is a **Float64** while `x[2]` is a **Int**, making that inner loop fast. In fact, it will only need to pay one *dynamic dispatch*, i.e. a multiple dispatch determination that happens at runtime. But inside it has static dispatch. Notice that whenever functions are inferred, the dispatching is static since the choice of the dispatch is already made and compiled into the LLVM IR.

Specialization at Compile Time

Julia code will specialize at compile time if it can prove something about the result. For example:

```
function fff(x)
    if x isa Int
        y = 2
    else
        y = 4.0
    end
end
```

```
x + y
end
```

You might think this function has a branch, but in reality Julia can determine whether **x** is an **Int** or not at **compile time**, so it will actually compile it away and just turn it into the function **x+2** or **x+4.0**:

```
@code_llvm fff(5)
```

```
@code_llvm fff(2.0)
```

Thus one does not need to worry about over-optimizing since in the obvious cases the compiler will actually remove all of the extra pieces when it can! So if have more type information that is possible to use then feel free to use. Note the code is still generic.

Global Scope and Optimizations

This discussion shows how Julia's optimizations all apply during function specialization times. **Thus calling Julia functions is fast.** But what about when doing something outside of the function, like directly in a module or in the REPL?

```
A = rand(100, 100)
B = rand(100, 100)
C = rand(100, 100)
@btime for j in 1:100, i in 1:100
    C[i,j] = A[i,j] + B[i,j]
end
```

This is very slow because the types of **A**, **B**, and **C** cannot be inferred. Why can't they be inferred? Well, at any time in the dynamic REPL scope I can do something like **C = "haha now a string!"**, and thus it cannot specialize on the types currently existing in the REPL (since asynchronous changes could also occur), and therefore it defaults back to doing a type check at every single function which slows it down.

But if we take the same code and does

```
function fmat(A, B, C)
    for j in 1:100, i in 1:100
        C[i,j] = A[i,j] + B[i,j]
    end
end

@btime fmat(A, B, C)
```

Moral of the story, Julia functions are fast but its global scope is too dynamic to be optimized.