Lesson 5, Week 2: Type System (as it relates to multiple dispatch)

AIM

— To describe enough of Julia's type system for explaining how Julia does multiple dispatch

After this lesson, you will be able to

* Describe how Julia's type system consists abstract and concrete types
* Use the `supertype` function on a type to find its supertype
* Use the `subtypes` function on a type to find its subtypes
* Describe the abstract type `Any`
* Explain how multiple dispatch matches the type signature of a function call to the type specification of a method

An illlustrative example

Let's go back to the function `double` defined in the previous lecture, with the two methods
`double(x:Int64) = 2x` and
`double(x::String) = x^2`
and recall that this throws an error for `double(1.2)` .

We could add a method for type `Float64`, but Julia offers us a different route: abstract types. In this case, we can use the type `Number`, as follows:

`double(x::Number) = 2x` DEMO: now `double(3.3) works`

In fact, we could even go further and provide a default method, with `double(x) = 2x`. This is a fall-back method, used only when the others do not apply. That is, it is used only when `x` is not `Int64`, nor `String`, nor any type that falls under the abstract type `Number`. The fall-back method accepts the value of `x` whatever its type may be. There is a way to specify this: the code `double(x::Any) = 2x` is exactly the same as `double(x) = 2x` .

Abstract versus concrete types

An abstract type is simply a way of talking about several types at the same time. If a Julia type has one or more subtypes, then it is an abstract type. We can check for this using the function `subtypes`. For example `Number` has the subtypes `Complex` and `Real`: [DEMO].

This means that although `Number` is a distant supertype of both `Int64` and `Float64`, it is not the immediate supertype of either of them. [DEMO: repeatedly use `subtype` to show that eventually the types `Int64` and `Float64` turn up.]

Every type in Julia is a subtype of exactly one type. We check for it using the function `supertype`.

DEMO:
verify that both `Complex` and `Real` have the supertype `Number`

On the other hand, some Julia types are *concrete types*: they have no subtypes.

DEMO: verify that `Int64`, `Float64` and `String` have no subtypes.

The most abstract type of all in Julia is `Any` which is its own supertype[1] and the eventual supertype of all Julia's types.

Here is a crucial point: every *value* in Julia has a concrete type. A variable does not in itself have a type. Sometimes we do speak of the type of a variable, for example for the function definition `double(x::Int64)` we might say that `x` is of type `Int64`, but that is simply for convenience[2].

What then of arrays? Julia, as far possible, tries to ensure that every element of an array has the same type. [DEMO: `[1, 1.0]` ends up with two values of type `Float64`.]

This makes for efficient computing[3], but it isn't always what we want. To allow arrays of mixed type, we explicitly have to declare the array to be of an abstract type that includes all the required types. The most general is of course `Any[...]`; less general but still abstract types can often be used.

[DEMO: `Number[1, 1.0]` is a convenient way to specify the required array[4].]

Julia's type system enables multiple dispatch

We can now answer the question: when a function has many methods, how does Julia choose which one to use? The answer lies in the type system, and in particular in the idea of subtypes.

Consider the call `double(7)`. The type signature of this call is "one variable of type `Int64`". There

[1]For completeness, we note that Julia's type system is a tree: if you start with any concrete type, and repeatedly ask for supertype, you always end up at `Any`.

[2]It is hopelessly pedantic to say "This method applies when the variable `x` has a value of type `Int64`", but that is what is accurate.

[3]The ability to ensure efficient computing is one of Julia's very strong points. It requires careful coding of functions to ensure type stability, but that is a topic beyond the scope of this course.

[4]For completeness, we note that there are other ways to do the same thing. We won't study them in this course.

are three methods which may apply, with type specifications `double(x::Int64)`, `double(x::Float64)` and `double(x::Any)`. Julia chooses the one with the more concrete type specification, in this case the method for `x::Int64`.

It works like this. Firstly, Julia chooses a method for a function only when it calls that function while it is actually running the code[5]. Since all values in Julia have a concrete type, the type signature of function call consists of concrete types. Secondly, Julia searches the methods for type specifications that match. For each value in the type signature of the call, it searches for a method that matches that value—if there is no method for the concrete type, it searches for a method with the supertype of that concrete type, if none such then for a method with the supertype of the supertype, and so on[6].

Ideally, the search described above finds exactly one method. Here is an example with two methods where it doesn't:

```
disptype(x::Int64, y) = println("x requires Int64, y can be Any")
disptype(x, y::Int64) = println("x can be Any, y requires Int64")
```

[DEMO: verify that `disptype(7, 7.0)` works, `disptype(7.0, 7.0)` has no method, `disptype(7.0, 7)` works, and `disptype(7, 7)` is flagged as ambiguous, with two possible methods.]

It is part of your job as a programmer to make sure all of your methods apply exactly where and when you want them to. On this course, that will be easy.

A final remark

Julia's type system includes thousands of types. Expert Julia programmers know very many of them. Moreover, in their code they often extend Julia's type system with their own user-defined types. It is a very powerful system and at the heart of what makes Julia a language that is easy to extend while still being very, very fast.

However, you need not specify type ever, in any of the code you write, and on this course only a few exercises ask you to do so. This makes code in Julia easy to write, even for a beginner[7], which is why this course is possible. You need to know about the type system for two reasons only: to help you with debugging code that throws type errors, and to understand how multiple dispatch works.

Review and summary

* `Number` is an abstract type

* The immediate supertype of a type is given by the function `supertype`

* The type `Number` is an eventual supertype of both `Int64` and `Float64`

[5]This is because Julia uses just-in-time compiling. This fascinating topic is beyond the scope of this course.

[6]This ends with the type `Any`, which includes all of Julia's types.

[7]The price is that you sometimes your code will not run anywhere near the maximum speed that Julia could deliver if types were carefully used.

* The type `Any` is the eventual supertype of all values in Julia

* The subtypes of a type is given by the function `subtypes`

* A concrete type has no subtypes

* Julia's values all belong to concrete types; all actual computations are done with concrete types

* Multiple dispatch is Julia's way of matching type specification of function method to type signature of function call; all the values in the call must match the type specification in the code.

* It is possible for a function call to be ambiguous, in the sense of matching more than one function specification.