Notes 3: Types and dispatch

Chris Paciorek

2025-01-28

Table of contents

Introduction

This document is the third of a set of notes, this document focusing on types and method dispatch based on types. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

Composite types (structs)

A struct is a collection of named fields, useful for holding information of a particular structure.

Here's a struct meant to contain information about a Gaussian process, with each element in the type also having its own declared type (though that is not required).

Objects of the type are constructed by calling the name of the type with the field values, in order.

```
struct GaussianProcess
  x::Vector
  covFun::Function
  params
end

function expCov(dists, params)
   return params[2]*exp.(-dists / params[1])
end

myGP = GaussianProcess([0:0.01:1;], expCov, (1, .5));
myGP.params
```

```
(1, 0.5)
```

It appears one has to know the exact order of the inputs to the constructor and give them by position rather than by name.

> ⚠ Naming of types
>
> Type names (including structs) are generally capitalized. One nice thing is that it helps distinguish constructors (e.g., `GaussianProcess`) from regular functions.

Struct constructors

We can create an explicit (*outer*) constructor that gives some flexibility in terms of what inputs the user can provide.

```
function ExpGaussianProcess(x, params)
   GaussianProcess(x, expCov, params)
end

myGP = ExpGaussianProcess([0:0.01:1;], (1, .5));
```

We can use an *inner* constructor to check inputs. This is a bad example because by defining types for the variables (as seen in the original definition of the `GaussianProcess` struct), this can be handled automatically by Julia.

```
struct GaussianProcess2
  x
  covFun
  params
  ## Define the constructor using function creation shorthand and if-else shorthand:
```

```
  GaussianProcess2(x, covFun, params) = isa(covFun, Function) ?
                    new(x, covFun, params) : error("covFun needs to be a function")
end

myGP = GaussianProcess2([0:0.01:1;], expCov, (1, .5));

myGP = GaussianProcess2([0:0.01:1;], 3, (1, .5))
```

```
LoadError: covFun needs to be a function
covFun needs to be a function

Stacktrace:
 [1] error(s::String)
   @ Base ./error.jl:35
 [2] GaussianProcess2(x::Vector{Float64}, covFun::Int64, params::Tuple{Int64, Float64})
   @ Main ./In[4]:6
 [3] top-level scope
   @ In[4]:12
```

> ⚠ Types can't be redefined
>
> We can't redefine a struct (hence my use of `GaussianProcess2` above). I think this has to do with the fact that this would break methods that have been specialized to the type.

Type declarations

Type declarations are a good and easy way to make our code robust without including a bunch of manual assertions.

Type declarations for functions

We can declare types for function arguments.

```
function mysum(x::Float64, y::Float64)
  return x+y
end

mysum(3.5, 4.8)
```

```
8.3
```

```
mysum(3, 4.8)
```

```
LoadError: MethodError: no method matching mysum(::Int64, ::Float64)
Closest candidates are:
  mysum(::Float64, ::Float64)
   @ Main In[5]:1
```

```
MethodError: no method matching mysum(::Int64, ::Float64)
Closest candidates are:
  mysum(::Float64, ::Float64)
   @ Main In[5]:1
Stacktrace:
 [1] top-level scope
   @ In[6]:1
```

```
mysum("hello", 4.8)
```

```
LoadError: MethodError: no method matching mysum(::String, ::Float64)
Closest candidates are:
  mysum(::Float64, ::Float64)
   @ Main In[5]:1
MethodError: no method matching mysum(::String, ::Float64)
Closest candidates are:
  mysum(::Float64, ::Float64)
   @ Main In[5]:1
Stacktrace:
 [1] top-level scope
   @ In[7]:1
```

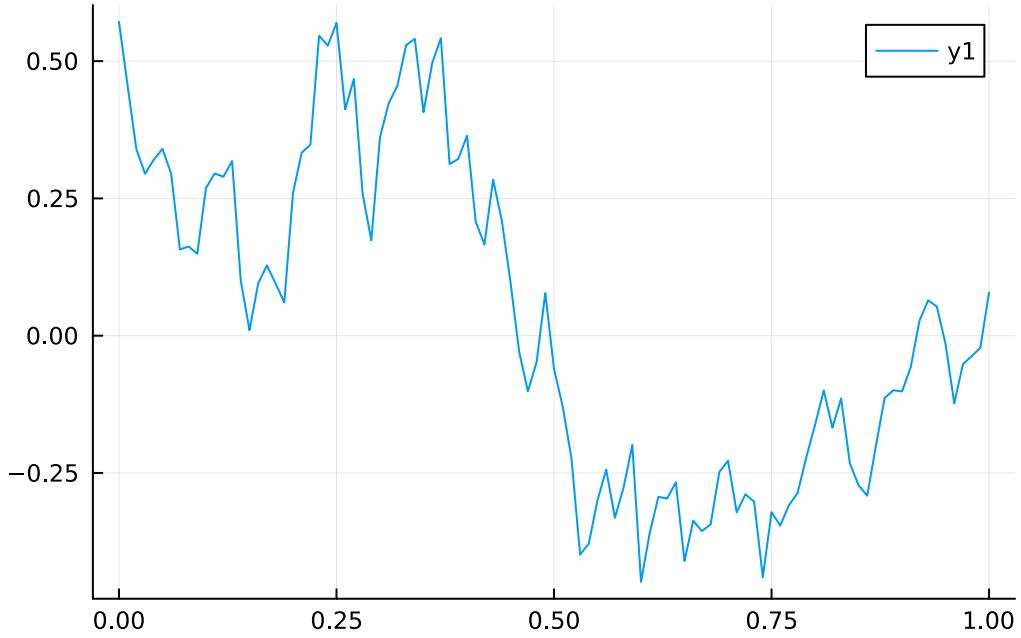
Type declarations with structs

```
using Random, Distributions, LinearAlgebra, Plots

function simulate(gp::GaussianProcess)
  n = length(gp.x)
  dists = abs.(gp.x .- gp.x')
  cov = gp.covFun(dists, gp.params)
  L = cholesky(cov).L
  y = L * rand(Normal(), n)
  return y
end

Random.seed!(123)
myGP = GaussianProcess([0:0.01:1;], expCov, (1, .5));
f = simulate(myGP);
```

```
plot(myGP.x, f)
```




```
tmp = simulate(7)
```

```
LoadError: MethodError: no method matching simulate(::Int64)
Closest candidates are:
  simulate(::GaussianProcess)
   @ Main In[8]:3
MethodError: no method matching simulate(::Int64)
Closest candidates are:
  simulate(::GaussianProcess)
   @ Main In[8]:3
Stacktrace:
 [1] top-level scope
   @ In[10]:1
```

Type unions and abstract types

```
IntOrFloat = Union{Float16, Float32, Float64, Int16, Int32, Int64}
function mysum(x::IntOrFloat, y::IntOrFloat)
  return x+y
end

mysum(3.5, 4.8)
```

```
8.3
```

```
mysum(3, 4.8)
```

```
7.8
```

```
typeof(mysum(3, 4))
```

```
Int64
```

```
mysum("hello", 4.8)
```

```
LoadError: MethodError: no method matching mysum(::String, ::Float64)
Closest candidates are:
  mysum(::Float64, ::Float64)
   @ Main In[5]:1
  mysum(::UnionFloat16, Float32, Float64, Int16, Int32, Int64, ::UnionFloat16, Float32, Float64, Int1
   @ Main In[11]:2
MethodError: no method matching mysum(::String, ::Float64)
Closest candidates are:
  mysum(::Float64, ::Float64)
   @ Main In[5]:1
  mysum(::UnionFloat16, Float32, Float64, Int16, Int32, Int64, ::UnionFloat16, Float32, Float64, Int1
   @ Main In[11]:2
Stacktrace:
 [1] top-level scope
   @ In[14]:1
```

Alternatively (and better as it's provided as part of the language), there is already a *abstract* type of ints and floats. Abstract types help with having a hierarchy of types (more later).

```
function mysum(x::Real, y::Real)
  return x+y
end

mysum(3.5, 4.8)
```

```
8.3
```

```
mysum(3, 4.8)
```

```
7.8
```

```
mysum(3, 4)
```

```
7
```

```
typeof(mysum(3, 4))
```

```
Int64
```

```
mysum("hello", 4.8)
```

```
LoadError: MethodError: no method matching mysum(::String, ::Float64)
Closest candidates are:
  mysum(::Float64, ::Float64)
   @ Main In[5]:1
  mysum(::UnionFloat16, Float32, Float64, Int16, Int32, Int64, ::UnionFloat16, Float32, Float64, Int1
   @ Main In[11]:2
  mysum(::Real, ::Real)
   @ Main In[15]:1
MethodError: no method matching mysum(::String, ::Float64)
Closest candidates are:
  mysum(::Float64, ::Float64)
   @ Main In[5]:1
  mysum(::UnionFloat16, Float32, Float64, Int16, Int32, Int64, ::UnionFloat16, Float32, Float64, Int1
   @ Main In[11]:2
  mysum(::Real, ::Real)
   @ Main In[15]:1
Stacktrace:
 [1] top-level scope
   @ In[19]:1
```

Type declarations for function output

```
function mysum2(x::Real, y::Real)::Float64
  return x+y
end

typeof(mysum2(3, 4))
```

```
Float64
```

What do you think this will happen if we do this?

```
function mysum3(x::Real, y::Real)::String
  return x+y
end

mysum3(3, 4)
```

Types in arrays

We've already seen that one can create arrays of heterogeneous elements, but one can't modify a homogeneous array in a way that would make it heterogeneous.

Exercise

- Experiment with arrays that contain missing values (via `missing`). What happens with computations?
- What seems to be the difference between `missing` and `nothing`?

More on structs and constructors

Here's a slightly more involved example of a struct that ties together some of what we've seen so far. (For use later on, we'll make it a *mutable* struct so we can modify the struct elements.)

```
mutable struct Person
  name::String
  age::Real
  function Person(name::String, age::Real)
    @assert(age < 130, "Age exceeds human lifespan.")
    new(name, age)
  end
end

biden = Person("Joe Biden", 82)
```

```
Person("Joe Biden", 82)
```

These invocations will both fail.

```
lincoln = Person("Abraham Lincoln", "hello")
```

```
LoadError: MethodError: no method matching Person(::String, ::String)
Closest candidates are:
  Person(::String, ::Real)
   @ Main In[21]:4
MethodError: no method matching Person(::String, ::String)
Closest candidates are:
  Person(::String, ::Real)
   @ Main In[21]:4
Stacktrace:
 [1] top-level scope
   @ In[22]:1
```

```
lincoln = Person("Abraham Lincoln", 200)
```

```
LoadError: AssertionError: Age exceeds human lifespan.
AssertionError: Age exceeds human lifespan.

Stacktrace:
 [1] Person(name::String, age::Int64)
   @ Main ./In[21]:5
```

```
 [2] top-level scope
   @ In[23]:1
```

Multiple dispatch

We can define different versions of functions or operators (these are called *methods*) that are used depending on the input types.

Functions and methods

The first time we define a method, it creates the generic function and initial method. When we define further methods, they are added to the existing function.

```
function test(x)
  println("In test, called with arbitrary input: ", x)
end

function test(x::String)
  println("In test, called with a string: ", x)
end

test(7)
```

```
In test, called with arbitrary input: 7
```

```
test("hello")
```

```
In test, called with a string: hello
```

It's called *multiple* dispatch because the dispatching can depend on multiple arguments and not just the first, which is not generally the case in other languages.

Operator overloading is multiple dispatch

Multiple dispatch is basically what is called "overloading" in other languages.

Let's illustrate multiple dispatch with our mutable `Person` struct.

```
import Base.+

function +(person::Person, incr::Real)
  person.age += incr
  return person
end

barack = Person("Barack Obama", 60)
barack + 3;
barack
```

```
Person("Barack Obama", 63)
```

Now we'll set up another `+` method for adding two `Persons`.

```
struct Partnership
  person1::Person
  person2::Person
  year_formed::Int
end

function +(personA::Person, personB::Person)::Partnership
  return Partnership(personA, personB, 1990)
end

michelle = Person("Michelle Obama", 60)

obamas = barack + michelle
```

```
Partnership(Person("Barack Obama", 63), Person("Michelle Obama", 60), 1990)
```

That's a decent illustration of the power of multiple dispatch but it's awkward to hard-code in the "1990" in this case. If we were using multiple dispatch with a function, then we wouldn't be restricted to having two arguments.

We've now added a couple more `+` methods to the large number already existing in Julia's `Base` module/library.

Note the similarity to object-oriented programming, but the methods are not part of classes.

> **Extensibility**
>
> This an example of *extensibility*. Julia's core functionality will work with user-defined objects.

Show methods

The same idea extends to other core Julia functions/functionality, such as printing objects.

Here we'll overload the Base `show` function. Before we do so, let's see what existing `show` methods there are:

```
Base.show
# methods(Base.show)

print(myGP)

function Base.show(io::IO, gp::GaussianProcess)
  println("A Gaussian process defined on a grid from $(gp.x[1]) to $(gp.x[end]).")
end

print(myGP)
myGP
```

```
GaussianProcess([0.0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.11, 0.12, 0.13, 0.
```

```
A Gaussian process defined on a grid from 0.0 to 1.0.
```

This is similar to combining the `__str__` and `__repr__` "dunder" (double underscore) methods for Python classes.

Subtyping

Types can relate to each other. Generally in Julia one groups types as subtypes of an *abstract* type whose purpose is to relate the subtypes. This is similar to class inheritance in object-oriented programming.

```
abstract type GeneralPerson end

mutable struct President <: GeneralPerson
    name::String
    age::Real
    inauguration_year::Int64
end

mutable struct Employee <: GeneralPerson
    name::String
    age::Real
    employer::String
end
```

The benefit is that we can define a single function/operator that works with multiple subtypes.

```
# These will work with both Employee and President
function +(person::GeneralPerson, incr::Real)
  person.age += incr
  return person
end

function Base.show(io::IO, person::GeneralPerson)
  println("A person of age $(person.age).")
end

macron = President("Emmanuel Macron", 48, 2017)
```

```
A person of age 48.
```

```
macron + 1
```

```
A person of age 49.
```

Well that worked as an illustration, but in this case it's not clear that `macron` is a special kind of person based on the printout. So we probably would want special `show` methods for the concrete types.

Abstract types vs. concrete types

The purpose of abstract types is to organize types into a hierarchy. In principle, one doesn't create instances of an abstract type (hence "abstract"), only of a concrete type, and yet, one can do `Real(4)` and `Number(4)`, even though those are abstract types, so I'm not fully understanding something here.

Type ordering

We can check if a type is a subtype like this:

```
Int64 <: Real
```

```
true
```

```
Real <: Number
```

```
true
```

```
Number <: Any
```

```
true
```

```
Employee <: GeneralPerson
```

```
true
```

```
Employee <: Float64
```

```
false
```

```
GeneralPerson >: Employee
```

```
true
```

> Use of `:`
>
> Note that `:` appears in various ways related to Julia code structure:
>
> ```
> x::Real        # type declaration
> Int64 <: Real  # type hierarchy
> :x             # representing a variable as code in meta programming
> ```

Nested subtypes

One can't have subtypes of non-abstract (concrete) types.

Here's what the Julia manual has to say: "It turns out that being able to inherit behavior is much more important than being able to inherit structure, and inheriting both causes significant difficulties in traditional object-oriented languages."

So the idea seems to be that there's not much point to having one type that inherits the fields (structure) of another type. Though I'm not sure what the difficulties are in inheriting both.

```
mutable struct Electrician <: Employee
    name::String
    age::Real
    employer::String
    times_electrocuted::Int
end
```

```
LoadError: invalid subtyping in definition of Electrician: can only subtype abstract types.
invalid subtyping in definition of Electrician: can only subtype abstract types.

Stacktrace:
 [1] top-level scope
   @ In[38]:1
```

Parametric types

There's even more flexibility in terms of types, if we parameterize types.

We can use `T` as a parametric type that accommodates various types (in this case only numeric ones).

This is not a particularly realistic example, but it enforces that both the `age` and `inauguration_year` have the same numeric type.

```
mutable struct USPresident{T <: Real} <: GeneralPerson
    name::String
    age::T
    inauguration_year::T
end

USPresident("Grant", 125, 1868)
```

```
A person of age 125.
```

```
USPresident("Grant", 125.3, 1868.0)
```

```
A person of age 125.3.
```

```
USPresident("Grant", 125.3, 1868);
```

```
LoadError: MethodError: no method matching USPresident(::String, ::Float64, ::Int64)
Closest candidates are:
  USPresident(::String, ::T, ::T) where T<:Real
   @ Main In[39]:2
MethodError: no method matching USPresident(::String, ::Float64, ::Int64)
Closest candidates are:
```

```
  USPresident(::String, ::T, ::T) where T<:Real
    @ Main In[39]:2
Stacktrace:
 [1] top-level scope
    @ In[41]:1
```