04. Type inference, type stability and so on

> *It is believed that if you want programmer productivity, you should use a **dynamic language**, such as Python, or R. On the other hand, if you want fast code execution, you should use a **statically typed language**, such as C or Java.*

- A language is statically-typed if the type of a variable is known at compile-time instead of at run-time.
- In Statically typed languages, once a variable has been declared with a type, it cannot ever be assigned to some other variable of different type and doing so will raise a type error at compile-time(some IDE's generally shows a Red Cross mark denoting the error).
- Static typing usually results in compiled code that executes more quickly because when the compiler knows the exact data types that are in use, it can produce optimized machine code (i.e. faster and/or using less memory).
- A language is **dynamically-typed** if the type of a variable is checked during run-time.
- In dynamically typed languages, variables are bound to objects at run-time by means of assignment statements, and it is possible to bind the same variables to objects of different types during the execution of the program.
- Dynamic type checking typically results in less optimized code than static type checking. It also includes the possibility of run time type errors and forces run time checks to occur for every execution of the program This is something that is especially prominent in scientific computing. This is the situation where the performance-critical inner kernel is written in C, but is then wrapped and used from a dynamic, higher-level language. Code written in traditional, scientific computing environments such as R, Matlab, or NumPy follows this paradigm.

Code written in this fashion is not without its drawbacks, however. Even though it looks like it gets you the best of both worlds—fast computation, while allowing the programmer to use a high-level language—this is a path full of hidden dangers. For one, someone will have to write the low-level kernel. So, you need two different skill sets. If you are lucky enough to find the low-level code in C for your project, you are fine. However, if you are doing anything new or original, or even slightly different from the norm, you will find yourself writing both C and a

high-level language. This will severely limit the number of contributors that your projects or research will get: to be really productive, those contributors really have to be familiar with two languages.

Secondly, when running code routinely written in two languages, there can be severe and unforeseen performance pitfalls. When you can drop down to C code quickly, everything is fine. However, if, for time reasons, effort, skill or changing requirements, you cannot write a performance-intensive part of your algorithm in C, you'll find your program taking hundreds or even thousands of times longer than you expected.

> *Julia is the first modern language to make a reasonable effort to solve the two-language problem. It is a high-level, dynamic language with powerful features that make for very productive programming. At the same time, code written in Julia usually runs very quickly, almost as quickly as code written in statically typed languages*

Julia's Type Inference and the Compiler

Julia is a Just In Time (JIT) compiled language, rather than an interpreted one in which the code in a high-level language is converted to machine code for execution on the CPU at runtime. This allows Julia to be dynamic, without having the overhead of interpretation. However, Julia is fast not because it is JIT compiled. *One can write very slow code in Julia but it's still JIT compiled! Instead, the reason why Julia is fast is because, roughly it is the combination of the following two ideas:*

- Type inference (this allows to write a generic code but still get the types correct)
- Type specialization in functions (picking the correct method so that you get C like performance!)

About C and Python

We spent quite a good amount of time explaining some stuffs at the core level of the computer, and sort of explained why everything needs to have a type in computer. Some languages are more explicit about types, while others try to hide the types from the user. A type tells the compiler how to store and **interpret** the memory of a value. When you have a slab of memory of bits, you have to know what the bits mean. Importantly, it will know *what to do for function calls*. If the code tells it to add two floating point numbers, it will send them as inputs to somewhere which will give instructions to carry out operations on floating-point numbers. However if the types are not known, it cannot be actually computed until the types are fully known, since otherwise it's impossible to interpret the memory and understand how to operate.

In languages like C, the programmer has to declare the types of variables in the program. Here is a **sample code for C**,

```
void add(double *a, double *b, double *c, size_t n){
  size_t i;
  for(i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
    // val = a[i] + b[i]; ## put them in stack
    // c[i] = val

  }
}
```

The types are known at compile time because the programmer tells what are the types of everything. In C, for compiler it is possible to know all of the different sizes because you tell it exactly what the types are for everything. Therefore, it's enabled to optimize with type information.

In many interpreted languages like Python, types are checked at runtime. So when we do the **same thing in Python**

```
a[1] = 2 ## it creates a ponter
b[1] = 4 ## it creates another pointer
a + b
```

when the addition occurs, the Python interpreter will check the object holding the values and ask it for its types, and use those types to know how to compute the **+** function. For this reason, the add function in Python is rather complex since it needs to decode and have a version for all primitive types! Not only is there runtime overhead checks in function calls due to not being explicit about types, there is also a memory overhead since it is impossible to know how much memory a value with take since that's a property of its type. Thus the Python interpreter cannot statically guerentee exact unchanging values for the size that a value would take in the stack, meaning that the variables are not stack-allocated. This means that every number ends up heap-allocated, which hopefully begins to explain why this is not as fast as C.

Julia's solution

Julia is somewhat of a hybrid. For C, if you say **a** is going you a vector 64-bit floating point numbers and try to put something that is 128-bit object it will just fail. But Python is in the whole other direction where it is equipped to change its type anytime because in any case whenever it's doing a function call it's going to check what the type it has, so it's not strictly necessary to have the type information at the begining. So it's okay to keep on changing your types around. That check will cost something but it will know how to handle these type

changes whereas the C function will not. Julia kind of falls in the middle. That is the tradeoff between flexibility vs. proper memory management. WHat happened in julia, Types in Julia are also optional when writing. For example, we can write the following

```
# code chunk 1
a = 2
b = 4
a + b
```

that looks same as Python version. When this code is running on REPL, it will do something similar to Python because it cannot know when you change the type of the variables beforehand. But here is what the ncie thing happened in Julia, before JIT compilation, Julia runs a **type inference algorithm which will try to find out that `a`** is an **`Int`**, and **`b`** is an **`Int`** before run time. Moreover the type of **`a+b`** is deduced as an **`Int`** based on type of the variables, again before run time. We can see the type inference step using the macro **`@code_typed`**

```
# code chunk 2
# we will use @code_typed macro

f1(x) = 2x


julia> f1(x) = 2x
# f1 (generic function with 1 method)

julia> @code_typed f1(2)
# CodeInfo(
# 1 ─ %1 = Base.mul_int(2, x)::Int64
# └──      return %1
# ) => Int64
```

The **`Int64`** is the important part, in this it immediately sees the types because it is easy to infer the types, and then it will specialize. So when a functions is invoked, Julia will check the types of the variable, but when it enters the function, it will start to deduce what's going on with the type of the objects it contains. Let's have a closer look on how type inference plays a role when a function is invoked.

Type Specialization in Functions

Now we can actually see what happens in the type specialization phase. All this means is we know the types now and then LLVM (low level virtual machine) compiler and specializes the code for a specific type

```
julia> @code_llvm f1(2)

# ;  @ REPL[5]:1 within `f1'
# define i64 @julia_f1_839(i64) {
# top:
# ; ┌ @ int.jl:87 within `*'
#    %1 = shl i64 %0, 1
# ; └
#   ret i64 %1
# }
```

Notice this only works for **64** bit integer. What happens if we give a floar

```
#  @code_llvm f1(2.0)

#  ;  @ REPL[5]:1 within `f1'
# define double @julia_f1_841(double) {
# top:
# ; ┌ @ promotion.jl:312 within `*' @ float.jl:405
#    %1 = fmul double %0, 2.000000e+00
# ; └
#   ret double %1
# }
```

Now it is for double. And this is **type specialization** in action. Now finally what we have seen yesterday,

```
foo(x , y) = x + y

julia> @code_llvm foo(2, 3.0)

# ;  @ REPL[11]:1 within `foo'
# define double @julia_foo_846(i64, double) {
# top:
# ; ┌ @ promotion.jl:311 within `+'
# ; │┌ @ promotion.jl:282 within `promote'
# ; ││┌ @ promotion.jl:259 within `_promote'
# ; │││┌ @ number.jl:7 within `convert'
# ; ││││┌ @ float.jl:60 within `Float64'
#        %2 = sitofp i64 %0 to double
# ; │└└└└
# ; │ @ promotion.jl:311 within `+' @ float.jl:401
#    %3 = fadd double %2, %1
# ; └
#   ret double %3
# }
```

Yes, it looks like it does a little bit more, but still there is no type checking here, it is, just like a handwritten C code, perfectly optimized for the specific types.

so each time that you change the types that are going to your function, there is a different function that's compiled. If we change the values keeping the types same, we have the same function.

Now restart the REPL and try

```
# code chunk 8
f(x,y) = x+y
@time @code_llvm foo(2.0,8.0)
@time @code_llvm foo(2.0,5.0)
```

So what happens when you call a function (with the same types) multiple times? First time it compiles, and then when you execute this for the second time (or maybe more), it already compiled this. So it just uses the same information again and runs it. So type information is a compile time information, and then the **actual value** of the floating point numbers is a runtime information you can specialize on the type information, but you cannot specialize the function on the runtime information and so the way that julia then works here is that because it uses type information, it makes things fast (similar to C). When you give a combination of functions, it will do the same thing, but will do it for one level more. Let's also check with **`@code_typed`**

```
@code_typed foo(2.0, 8.0)
```

You see **`Float64`** so it automatically recognizes.

Now, if it *cannot figure out the type* in that case what we have is called **type-instability**. We can use the **`@code_warntype`** macro to better see the inference along the steps of the function:

```
foo(x , y) = x + y
@code_warntype foo(2,5.0)
@code_warntype foo(2, 5)
```

This is a very useful macro to detect the type-instability in your code.

Type Stability

When successful type inference happens at compile time, we call the function a **`type-stable`** function. Does type inference always happen smoothly? Or more importantly we may

ask when does it fail? Here is an example,

```
function h(x, y)
    out = x + y
    rand() < 0.5 ? out : Float64(out)
end


julia> @code_warntype h(2,5)
# Variables
#   #self#::Core.Compiler.Const(h, false)
#   x::Int64
#   y::Int64
#   out::Int64

# Body::Union{Float64, Int64}
# 1 ─      (out = x + y)
# │   %2 = Main.rand()::Float64
# │   %3 = (%2 < 0.5)::Bool
# └──      goto #3 if not %3
# 2 ─      return out
# 3 ─ %6 = Main.Float64(out)::Float64
# └──      return %6
```

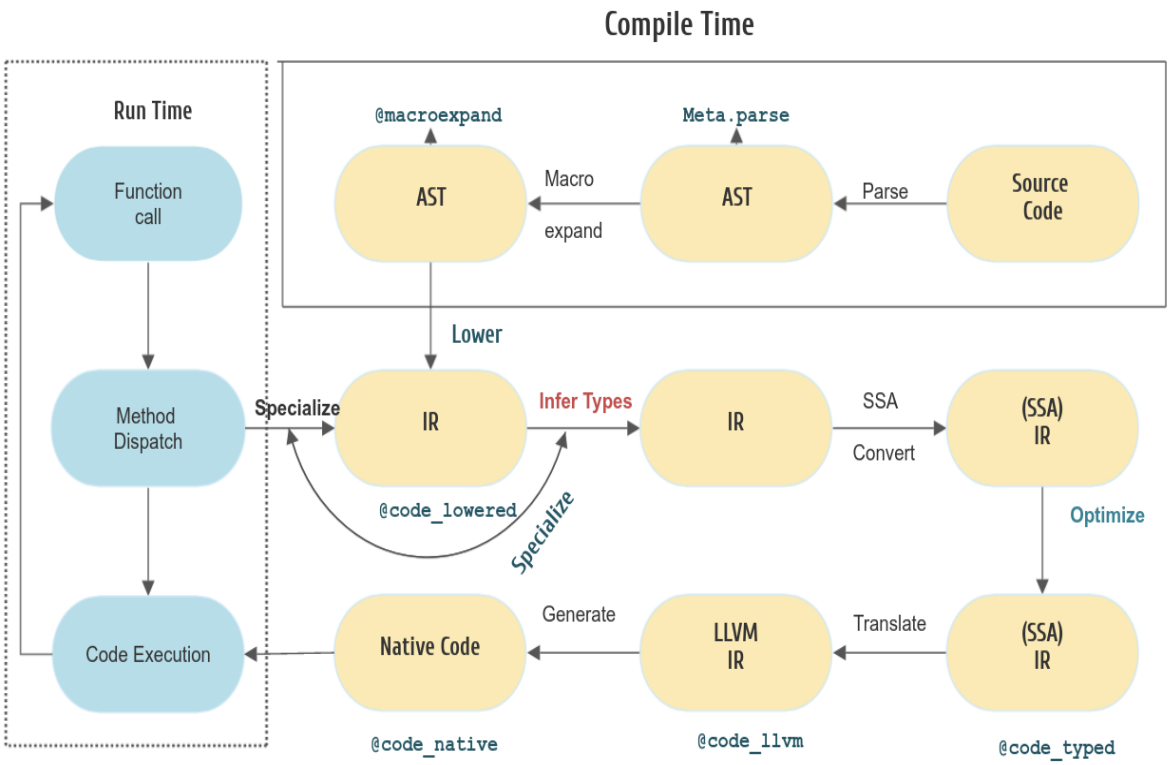
Look at the thing in red. Here, on an integer input the output's type is randomly either **`Int`**or **`Float64`**, and thus the output is unknown at compile time. How can we change this function to a **`type-stable`** (and also another function), maybe we write **`1 < 2`** in place of **`rand() < 0.5`**

```
# code chunk 17
function h2(x,y)
    out = x + y
    1 < 2 ? out : Float64(out)
end

julia> @code_warntype h2(2,5)
# Variables
#   #self#::Core.Compiler.Const(h2, false)
#   x::Int64
#   y::Int64
#   out::Int64

# Body::Int64
# 1 ─      (out = x + y)
# │   %2 = (1 < 2)::Core.Compiler.Const(true, false)
# │        %2
# └──      return out
# 2 ─      Core.Compiler.Const(:(Main.Float64(out)), false)
# └──      Core.Compiler.Const(:(return %5), false
```

Looks fine, can we see something from `@code_llvm` for the type unstable function.

Compilation pipeline

Now we can try to understand the compilation pipleline, and hopefully things will make more sense.



What Lionel explained is why some people say that Julia has a just ahead of time compiler. So let's try to see what this means with the compilation pipeline of Julia. So when we are executing a Julia code,

- It all begins with the source code,
- Source code is parsed and then macro expanded and this is all done from an AST (Abstract Syntax Tree, lets see some picture) which is a Julia object of type **`exp`** (recall expression)
- Then this expression is lowered to an internal representation (IR) and as you can see, that we can access the different stages of compilation using different macros.
- Once code has been loaded into memory runtime can begins and runtime consists in a very simple loop,
- You have a function call and this function is **dispatched** and then you get a method which is executed and which will probably call another method and so on.

- Now the interesting part is what happens during the dispatch.
- Julia **specializes the method you have dispatched on according to the concrete types of all of the arguments**.
- So when you're dispatching either you have a concrete call where you already have in memory a specialized method for this particular call will be executed.
- Otherwise you begin with a specialization cycle.
- Specialization takes the internal representation that was loaded in memory of the method it was dispatched on and begins with **type inference** which is a *dataflow analysis that propagates the types so the concrete types from those from the signature through the body of the function*.
- What can happen during this stage is that maybe you have another call within this method you're compiling and if type inference is good enough in your case you may be able to infer the concrete types of all of the arguments to this new call.
- In that case you already know **at compile time what method this particular function call will be dispatched on**.
- As a consequence you can already launch another specialization cycle, this is the reason for the back arrow here.
- Once code has been typed, after than it goes through a series transformation.
- Finally after code is optimized it is translated to LLVM that generates native code
- Once we have native code it is stored in memory for in the method table of the function so that next time the same function call the function call with the same causing signature happens it can be directly the specialized

Let's see an example of a type-stable code,

```
function intlog2(n) # n::Int64
    r = 0
    # n::Int64
    while n > 1
        # n::Int64
        r += 1
        # n::Int64
        n = n/2
        # n::Float64
    end
```

```
    return r
end
```

2nd loop

```
function intlog2(n) # n::Int64
    r = 0
    while n > 1
        # n::Union{Int64, Float64}
        r += 1
        # n::Union{Int64, Float64}
        n = n/2
        # n::Union{Int64, Float64}
    end
    return r
end
```

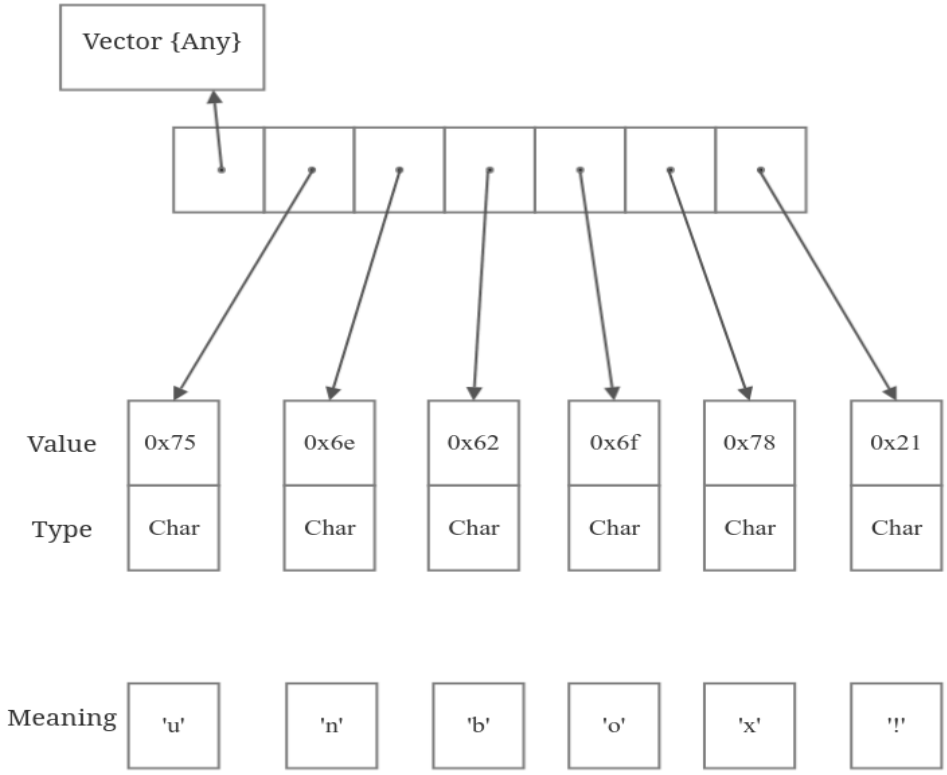
the type of **n** outside the loop will be **Union{Int64, Float64}** because it is possible that we never entered the loop. All the variables are also inferred in the type inference, so we will get

```
function intlog2(n) # n::Int64
    r = 0
    while n > 1
        # n::Union{Int64, Float64}, r::Int64
        r += 1
        # n::Union{Int64, Float64}, r::Int64
        n = n/2
        # n::Union{Int64, Float64}, r::Int64
    end

    return r # r::Int64
end
```

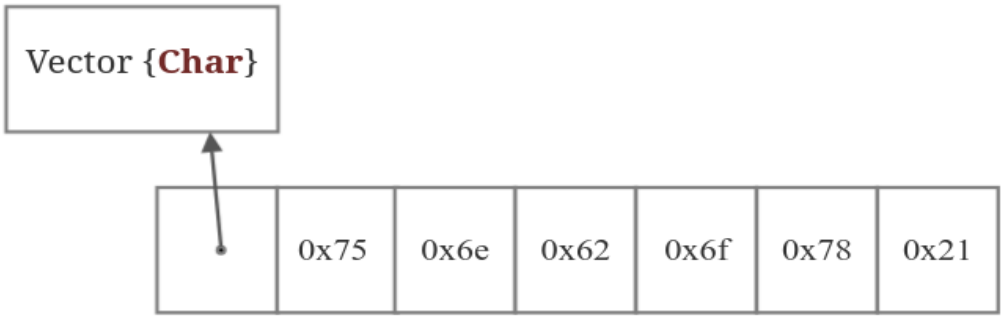
So at the end we know that return type of this method will be always **Int64**. As a consequence this is what we say a **type-stable** method. Meaning compiler could perfectly infer the types. If this function is called again, it will just reuse this information.

Boxing and Unboxing

This is from Steve Johnson's lecture and also from Lionel's presentation.



You are just dead. Julia unboxes



Why this is useful. This explains why julia is faster for loops rather than vectorization.

References

- There is wonderful lecture series from Chris Rackaukas, Youtube link. Click on the link. The course also has many other things which you can use to learn if you are interested. I learned many things from here.

- Presentation of Lionel Zoubritzky at JuliaCon. I think the very first presentation which went to this depth.

- Steve Johonson's presentation for the course - MIT 6.172 Performance Engineering of Software Systems, Fall 2018. The course is also a fantastic course, but more towards theory, here is the page. All video lectures are available at MIT OCW.

- Magic lies here - Statically vs Dynamically Typed Languages . There are 100s of article like this. But I must admit this was good. So I added some stuffs from here.

- Sengupta, A., & Edelman, A. (2019). Julia High Performance: Optimizations, distributed computing, multithreading, and GPU programming with Julia 1.0 and beyond, 2nd Edition. Packt Publishing.

- Kwong, T., & Karpinski, S. (2020). Hands-On Design Patterns and Best Practices with Julia: Proven solutions to common problems in software design for Julia 1.x. Packt Publishing.

I think the above two books by Packt I really liked and read some parts. Probably I will take some parts from there for the coming sections as well. Problems is Julia changed so much so many books have become outdated, so these are relatively the recent ones and also quite well written. I am sorry honestly, even if I didn't want to, some of the references I unintentionally missed, which happened out of hurry. But I should be more careful as I mentioned, **this is serious stuff!**. So I better be more careful. But if you think, if I missed anything and you are able catch me, feel free to let me know, I will consider this as a favor. Since I honestly believe this is a bad practice, I will be really happy. The people should be recognized for what they have done. But finally I hope now it is ok, and I am not missing my references.