Some Julia specifics

Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 04
 Jürgen Fuhrmann

```
begin
using PlutoUI
using LinearAlgebra
using InteractiveUtils
import AbstractTrees
end
```

Table of Contents

Type system basics

- Julia is a strongly typed language
- Knowledge about the layout of a value in memory is encoded in its type
- Prerequisite for performance

- There are concrete types and abstract types
- See the Julia WikiBook for more

Concrete types

- Every value in Julia has a concrete type
- Concrete types correspond to computer representations of objects
- Inquire type info using `typeof()`

Int64

```
typeof(10)
```

Float64

```
typeof(10.0)
```

ComplexF64 (alias for Complex{Float64})

```
typeof(3.0+3im)
```

Bool

```
typeof(false)
```

String

```
typeof("false")
```

Vector{Float16} (alias for Array{Float16, 1})

```
typeof(Float16[1,2,3])
```

Matrix{Int64} (alias for Array{Int64, 2})

```
typeof(rand(Int,3,3))
```

Structs

- Structs allow to define custom types

```
struct MyColor64
    r::Float64
    g::Float64
    b::Float64
end
```

c = MyColor64(0.1, 0.2, 0.3)

```
c=MyColor64(0.1,0.2,0.3)
```

MyColor64

```
1 typeof(c)
```

24

```
1 sizeof(c)
```

0.1

```
1 c.r
```

Error message from Main

> setfield!: immutable struct of type MyColor64 cannot be changedsetfield!: immutable struct of type MyColor64 cannot be changed

Stack trace

Here is what happened, the most recent locations are first:

1. setproperty!(x::MyColor64, f::Symbol, v::Float64)
   from julia → Base.jl:53
2. from This cell: line 1

```
1 c.r=0.5
```

```
1 c.r=0.5
```

Mutable structs

- allow changing fields

```
1 mutable struct MMyColor64
2      r::Float64
3     g::Float64
4     b::Float64
5 end
```

mc = MMyColor64(0.1, 0.2, 0.3)

```
1 mc=MMyColor64(0.1,0.2,0.3)
```

0.7

```
1 mc.r=0.7
```

```
MMyColor64(0.7, 0.2, 0.3)
```

```
mc
```

(note that in this case Pluto's reactivity is tricked out ...)

Type parametrized structs

Structs can be parametrized with types. This is similar to array types which are parametrized by their element types, and to C++ template classes.

```
struct MyColor{T}
    r::T
    g::T
    b::T
end
```

```
c2 =  MyColor(4.0f0, 25.0f0, 233.0f0)
```

```
c2=MyColor{Float32}(4.0,25,233)
```

```
MyColor(1, 2, 3)
```

```
MyColor(1,2,3)
```

```
MyColor{Float32}
```

```
typeof(c2)
```

Abstract types

- Abstract types label concepts which work for a several concrete types without regard to their memory layout etc.
- All variables with concrete types corresponding to a given abstract type (should) share a common interface
- A common interface consists of a set of functions with methods working for all types exhibiting this interface
- The functionality of an abstract type is implicitely characterized by the methods working on it
- This concept is close to "duck typing": use the "duck test" — "If it walks like a duck and it quacks like a duck, then it must be a duck" — to determine if an object can be used for a particular purpose
- When trying to force a parameter to have an abstract type,it ends up with having a conrete type which is compatible with that abstract type

The type tree

- Types can have subtypes and a supertype
- Concrete types are the leaves of the resulting type tree
- Supertypes are necessarily abstract
- There is only one supertype for every (abstract or concrete) type
- Abstract types can have several subtypes

```
[BigFloat, BFloat16, Float16, Float32, Float64]
```

```
subtypes(AbstractFloat)
```

```
[]
```

```
subtypes(Float64)
```

```
AbstractFloat
```

```
supertype(Float64)
```

```
Real
```

```
supertype(AbstractFloat)
```

```
Number
```

```
supertype(Real)
```

```
Any
```

```
supertype(Number)
```

- "Any" is the root of the type tree and has itself as supertype

```
Any
```

```
supertype(Any)
```

We can use the `AbstractTrees` package to visualize the type tree. We just need to define what it means to have children for a type.

```
AbstractTrees.children(x::Type) = subtypes(x)
```

```
AbstractTrees.print_tree(Number)
```

```
Number
├─ MultiplicativeInverse
│  ├─ SignedMultiplicativeInverse
│  └─ UnsignedMultiplicativeInverse
├─ Complex
└─ Real
   ├─ AbstractFloat
   │  ├─ BigFloat
   │  ├─ BFloat16
   │  ├─ Float16
   │  ├─ Float32
   │  └─ Float64
   ├─ AbstractIrrational
   │  └─ Irrational
   ├─ FixedPoint
   │  ├─ Fixed
   │  └─ Normed
   ├─ Integer
   │  └─ Bool
```

Functions, Methods and Multiple Dispatch

- Functions can have different variants of their implementation depending on the types of parameters passed to them.
- These variants are called **methods**
- All methods of a function `f` can be listed by calling `methods(f)`
- The act of figuring out which method of a function to call depending on the type of parameters is called **multiple dispatch**.

```
test_dispatch(x)="general case: $(typeof(x)), x=$(x)";
```

```
test_dispatch(x::Float64)="special case Float64, x=$(x)";
```

```
test_dispatch(x::Int64)="special case Int64, x=$(x)";
```

```
"special case Int64, x=3"
```

```
test_dispatch(3)
```

```
"general case: Bool, x=false"
```

```
test_dispatch(false)
```

```
"special case Float64, x=3.0"
```

```
test_dispatch(3.0)
```

Multiple dispatch

Here we defined a generic method which works for any variable type passed. In the case of

`Int64`, `AbstractFloat` or `Float64` parameters, special cases are handeld by different methods of the same function. The compiler decides which method to call. This approach allows to specialize implementations dependent on data types, e.g. in order to optimize perfomance.

The `methods` function can be used to figure out which methods of a function exists.

3 methods for generic function **test_dispatch** from Main.var"workspace#5":

- test_dispatch(x::**Int64**) in Main.var"workspace#5" at /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#0cc7808a-0955-11eb-0b4d-ff491af88cf5:1
- test_dispatch(x::**Float64**) in Main.var"workspace#5" at /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#0468c2da-0955-11eb-271b-5d84d5d8343d:1
- test_dispatch(x) in Main.var"workspace#5" at /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#f5cc25e6-0954-11eb-179b-eddff99dd392:1

```
1 methods(test_dispatch)
```

Operators for testing type relationships

`true`

```
1  Float64<: Number
```

`false`

```
1  Float64<: Integer
```

`false`

```
1 isa(3,Float64)
```

`true`

```
1 isa(3.0,Float64)
```

Abstract types for method dispatch

`dispatch2 (generic function with 2 methods)`

```
1 begin
2     dispatch2(x::AbstractFloat)="$(typeof(x)) <:AbstractFloat, x=$(x)"
3     dispatch2(x::Integer)="$(typeof(x)) <:Integer, x=$(x)"
4 end
```

`"Bool <:Integer, x=false"`

```
1 dispatch2(false)
```

`"Float16 <:AbstractFloat, x=13.0"`

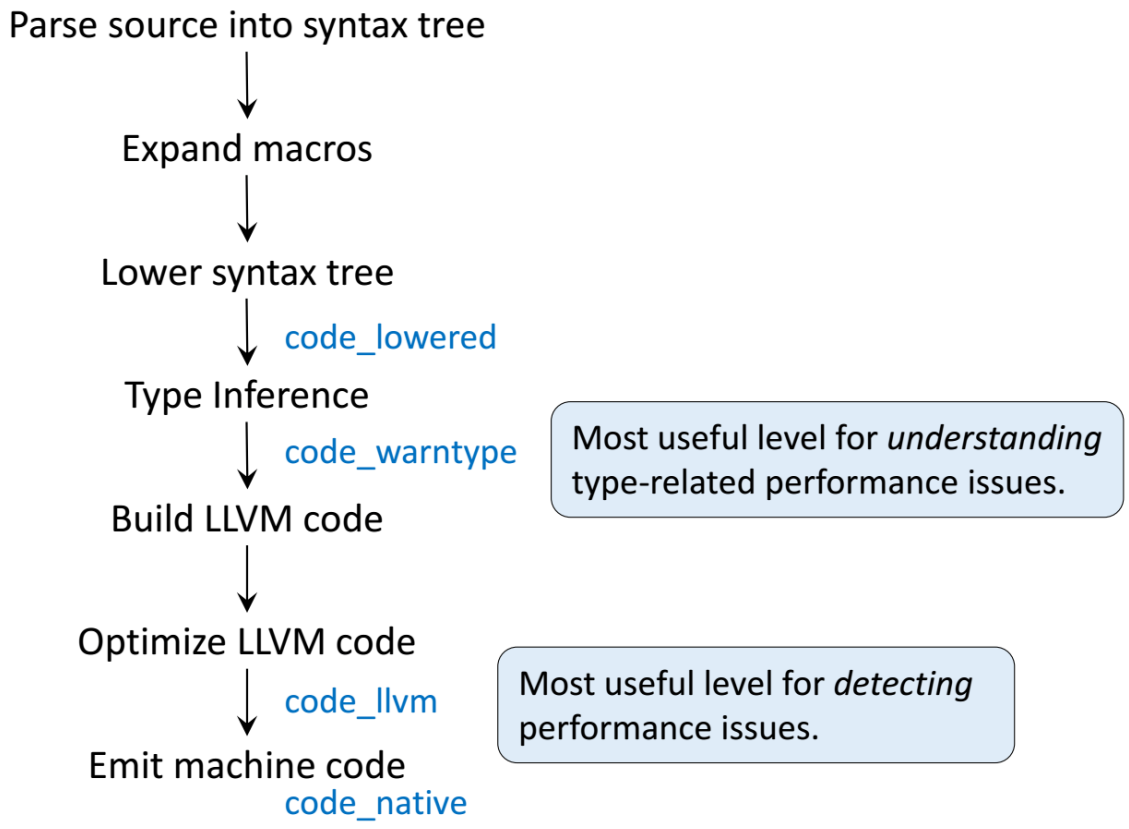
```
1 dispatch2(Float16(13.0))
```

The power of multiple dispatch

- Multiple dispatch is one of the defining features of Julia
- Combined with the the hierarchical type system it allows for powerful generic program design
- New datatypes (different kinds of numbers, differently stored arrays/matrices) work with existing code once they implement the same interface as existent ones.
- In some respects C++ comes close to it, but for the price of more and less obvious code

Compilation pipeline

- Just-in-time (JIT) or, more precisely, just-ahead of time (JAOT) compilation is another feature setting Julia apart
- Julia development started with this idea
- Julia uses the tools from the The LLVM Compiler Infrastructure Project to organize compilationof Julia code to machine code
- Tradeoff: startup time for code execution in interactive situations, however, since v1.9, machine code is cached on the disk for later reuse.
- Multiple steps: Parse the code, analyze data types etc.
- Intermediate results can be inspected using a number of macros (blue color in the diagram below)



From Introduction to Writing High Performance Julia by D. Robinson

Specialization of generic functions

```
g (generic function with 1 method)
```

```
1 g(x,y)=x+y
```

Call with integer parameter:

```
5
```

```
1 g(2,3)
```

Call with floating point parameter:

```
5.0
```

```
1 g(2.0,3.0)
```

@code_lowered

The macro `@code_lowered` describes the abstract syntax tree behind the code

```
CodeInfo(
1 ─ %1 = x + y
└──      return %1
)
```

```
1 @code_lowered g(2,3)
```

```
CodeInfo(
1 ─ %1 = x + y
└──      return %1
)
```

```
1 @code_lowered g(2.0,3.0)
```

@code_warntype

`@code_warntype` (with output to terminal) provides the result of type inference (detection of the parameter types and cooresponding choice of the translation strategy) according to the input:

```
1 @code_warntype g(2,3)
```

```
MethodInstance for Main.var"workspace#5".g(::Int64, ::Int64)
  from g(x, y) @ Main.var"workspace#5" ~/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1
Arguments
  #self#::Core.Const(Main.var"workspace#5".g)
  x::Int64
  y::Int64
Body::Int64
1 ─ %1 = (x + y)::Int64
└──      return %1
```

```
1 @code_warntype g(2.0,3.0)
```

```
MethodInstance for Main.var"workspace#5".g(::Float64, ::Float64)
  from g(x, y) @ Main.var"workspace#5" ~/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1
Arguments
  #self#::Core.Const(Main.var"workspace#5".g)
  x::Float64
  y::Float64
Body::Float64
1 ─ %1 = (x + y)::Float64
└──      return %1
```

@code_llvm

`@code_llvm` prints a human readable rendering og the LLVM intermediate byte code representation:

```
1 @code_llvm g(2,3)
```

```
; Function Signature: g(Int64, Int64)
;  @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1 within `g`
define i64 @julia_g_18133(i64 signext %"x::Int64", i64 signext %"y::Int64")
#0 {
top:
; ┌ @ int.jl:87 within `+`
   %0 = add i64 %"y::Int64", %"x::Int64"
   ret i64 %0
; └
}
```

```
1 @code_llvm g(2.0,3.0)
```

```
; Function Signature: g(Float64, Float64)
;  @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c75
c8c-c0f4-4800-bdab-cbaf17b55bb7:1 within `g`
define double @julia_g_18246(double %"x::Float64", double %"y::Float64") #0
{
top:
; ┌ @ float.jl:491 within `+`
   %0 = fadd double %"x::Float64", %"y::Float64"
   ret double %0
; └
}
```

@code_native

Finally, `@code_native` prints the assembler code generated, which is a close match to the machine code sent to the CPU:

```
1 @code_native g(2,3)
```

```
    .text
    .file   "g"
    .globl  julia_g_18358                   # -- Begin function julia_g_183
58
    .p2align    4, 0x90
    .type   julia_g_18358,@function
julia_g_18358:                          # @julia_g_18358
; Function Signature: g(Int64, Int64)
; ┌ @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c75
0c8c-c0f4-4800-bdab-cbaf17b55bb7:1 within `g`
# %bb.0:                                # %top
; │ @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c75
0c8c-c0f4-4800-bdab-cbaf17b55bb7 within `g`
    #DEBUG_VALUE: g:x <- $rdi
    #DEBUG_VALUE: g:y <- $rsi
    push    rbp
    mov rbp, rsp
; │ @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c75
```

```
1 @code_native g(2.0,3.0)
```

```
    .text
    .file   "g"
    .globl  julia_g_18456                   # -- Begin function julia_g_184
56
    .p2align    4, 0x90
    .type   julia_g_18456,@function
julia_g_18456:                              # @julia_g_18456
; Function Signature: g(Float64, Float64)
; ┌ @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c75
0c8c-c0f4-4800-bdab-cbaf17b55bb7:1 within `g`
# %bb.0:                                    # %top
; │ @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c75
0c8c-c0f4-4800-bdab-cbaf17b55bb7 within `g`
    #DEBUG_VALUE: g:x <- $xmm0
    #DEBUG_VALUE: g:y <- $xmm1
    push    rbp
    mov rbp, rsp
; │ @ /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c75
```

We see that for the very same function, Julia creates different variants of executable code depending on the data types of the parameters passed. In certain sense, this extends the multiple dispatch paradigm to the lower level by automatically created methods.

Method instances

```
1 using MethodAnalysis
```

```
Core.MethodInstance[
    1:  MethodInstance for Main.var"workspace#5".g(::Int64, ::Int64)
    2:  MethodInstance for Main.var"workspace#5".g(::Float64, ::Float64)
]
```

```
1 methodinstances(g)
```

1 method for generic function **g** from Main.var"workspace#5":

- g(x, y) in Main.var"workspace#5" at /home/fuhrmann/Wias/teach/adscicomp/pluto/nb04-julia-types.jl#==#2c750c8c-c0f4-4800-bdab-cbaf17b55bb7:1

```
1 methods(g)
```