Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 05

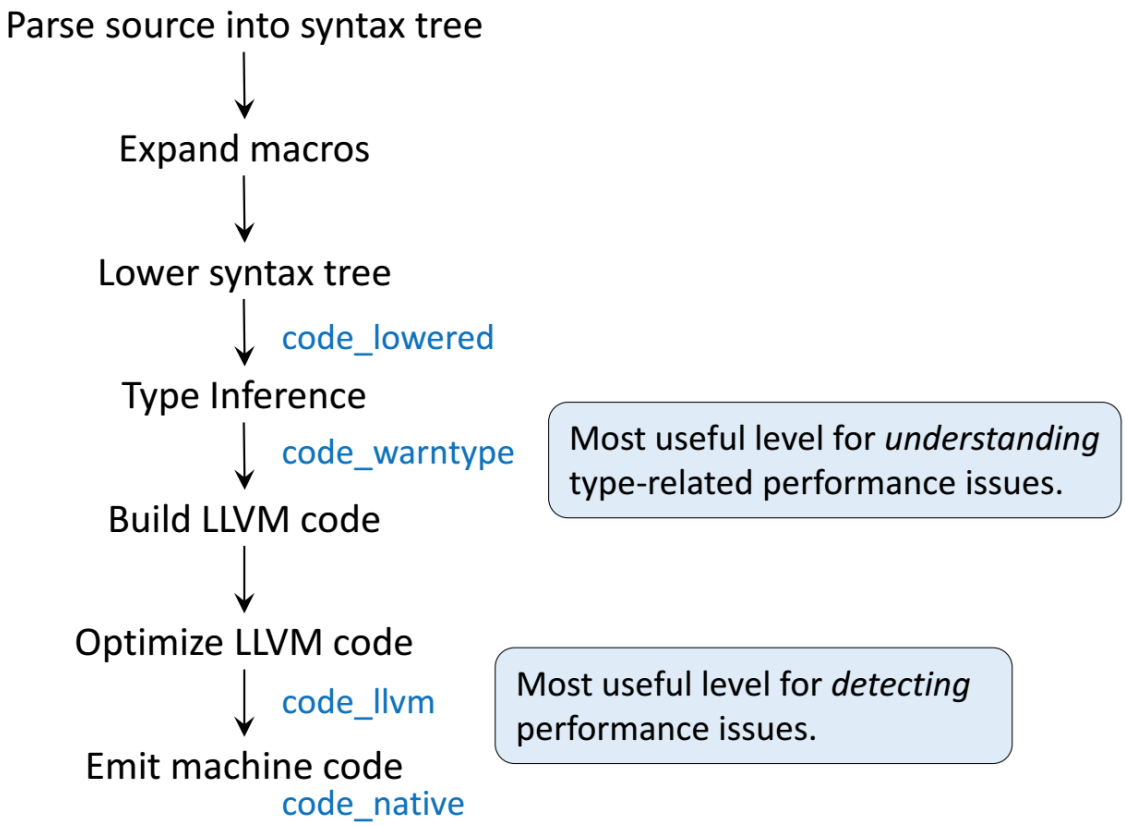
```
begin
    using PlutoUI
    using BenchmarkTools
    using LinearAlgebra
end
```

Julia: just-in-time compilation and Performance

Julia: just-in-time compilation and Performance

The JIT

- Just-in-time compilation is another feature setting Julia apart, as it was developed with this possibility in mind.
- Julia uses the tools from the **The LLVM Compiler Infrastructure Project** to organize on-the-fly compilation of Julia code to machine code
- Tradeoff: startup time for code execution in interactive situations
- 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

Let us see what is going on:

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

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

- Call with integer parameter:

```
5
```

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

- Call with floating point parameter:

```
5.0
```

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

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

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

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

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

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

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

```
Variables
  #self#::Core.Const(Main.workspace2.g)
  x::Int64
  y::Int64

Body::Int64
1 — %1 = (x + y)::Int64
└──      return %1
```

```
with_terminal() do
    @code_warntype g(2,3)
end
```

```
Variables
  #self#::Core.Const(Main.workspace2.g)
  x::Float64
  y::Float64

Body::Float64
1 — %1 = (x + y)::Float64
└──      return %1
```

```
with_terminal() do
    @code_warntype g(2.0,3.0)
end
```

- `@code_llvm` prints the LLVM intermediate byte code representation:

```
;  @ /home/fuhrmann/Wias/teach/scicomp/pluto/nb05-julia-jit.jl#==#ecb14696-01dc-1
define i64 @julia_g_2978(i64 signext %0, i64 signext %1) {
top:
; ┌ @ int.jl:87 within `+'
   %2 = add i64 %1, %0
; └
  ret i64 %2
}
```

```
with_terminal() do
    @code_llvm g(2,3)
end
```

```
;  @ /home/fuhrmann/Wias/teach/scicomp/pluto/nb05-julia-jit.jl#==#ecb14696-01dc-1
define double @julia_g_3020(double %0, double %1) {
top:
; ┌ @ float.jl:326 within `+'
   %2 = fadd double %0, %1
; └
  ret double %2
}
```

```
with_terminal() do
    @code_llvm g(2.0,3.0)
end
```

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

```
    .text
; ┌ @ nb05-julia-jit.jl#==#ecb14696-01dc-11eb-2c33-7f0c5f3ed551:1 within `g'
; │┌ @ int.jl:87 within `+'
    leaq    (%rdi,%rsi), %rax
; │└
    retq
    nopw    %cs:(%rax,%rax)
; └
```

```
with_terminal() do
    @code_native g(2,3)
end
```

```
    .text
; ┌ @ nb05-julia-jit.jl#==#ecb14696-01dc-11eb-2c33-7f0c5f3ed551:1 within `g'
; │┌ @ float.jl:326 within `+'
    vaddsd  %xmm1, %xmm0, %xmm0
; │└
    retq
    nopw    %cs:(%rax,%rax)
; └
```

```
with_terminal() do
    @code_native g(2.0,3.0)
end
```

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.

Performance measurment

- Julia provides a number of macros to support performance testing.
- Performance measurement of the first invocation of a function includes the compilation step. If in doubt, measure timing twice.
- Pluto has the nice feature to indicate the execution time used below the lower right corner of a cell. There seems to be also some overhead hidden in the pluto cell handling which is however not measured.

- `@elapsed` : wall clock time used returned as a number.

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

```
f(n1,n2)= mapreduce(x->norm(x,2),+,[rand(n1) for i=1:n2])
```

```
0.001590306
```

```
@elapsed f(1000,1000)
```

- `@allocated` : sum of memory allocated (including temporary) during the excution of the code. For storing intermediate and final calculation results, computer languages request memory from the operating system. This process is called allocation. Allocations as a rule are linked with lots of bookkeeping, so they can slow down code.

```
8136128
```

```
@allocated f(1000,1000)
```

- `@time` : `@elapsed` and `@allocated` together, with output to the terminal. Be careful to time at least twice in order to take into account compilation time. In addition, the number of allocations is printed along with time spent for garbage collection. Garbage collection is the process of returning unused (temporary) memory to the system.

```
18248.454341186032

  0.001729 seconds (1.00 k allocations: 7.759 MiB)
```

```
with_terminal() do
    @time f(1000,1000)
end
```

- `@benchmark` from `BenchmarkTools.jl` creates a statistic over multiple samples in order to give a more reliable estimate.

```
BenchmarkTools.Trial: 2663 samples with 1 evaluation.
 Range (min … max):  1.412 ms …   4.877 ms  ┊ GC (min … max):  0.00% … 39.52%
 Time  (median):     1.586 ms               ┊ GC (median):     0.00%
 Time  (mean ± σ):   1.872 ms ± 612.609 μs  ┊ GC (mean ± σ):  10.31% ± 13.95%
```

```
 1.41 ms      Histogram: log(frequency) by time      3.93 ms <

 Memory estimate: 7.76 MiB, allocs estimate: 1001.
```

```
@benchmark f(1000,1000)
```

Some performance gotchas

In order to write efficient Julia code, a number recommendations should be followed.

Gotcha #1: global variables

```
myvec =
  [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1
```

```
myvec=ones(Float64,1_000_000)
```

```
function mysum(v)
    x=0.0
    for i=1:length(v)
        x=x+v[i]
    end
    return x
end;
```

```
0.007108578
```

```
@elapsed mysum(myvec)
```

```
0.123235956
```

```
@elapsed begin
    x=0.0
    for i=1:length(myvec)
        global x
        x=x+myvec[i]
    end
end
```

- Observation: both the begin/end block and the function do the same operation and calculate the same value. However the function is faster.
- The code within the begin/end clause works in the *global context*, whereas in `myfunc`, it works in the scope of a function. Julia is unable to dispatch on variable types in the global scope as they can change their type anytime. In the global context it has to put all variables into "boxes" tagged with type information allowing to dispatch on their type at runtime (this is by the way the default mode of Python). In functions, it has a chance to generate specific code for known types.
- This situation als occurs in the REPL.

- Conclusion: **Avoid Julia Gotcha #1 by wrapping time critical code into functions and avoiding the use of global variables.**
- In fact it is anyway good coding style to separate out pieces of code into functions

Gotcha #2: type instabilities

```
f1 (generic function with 1 method)
```

```
function f1(n)
  x=1
  for i = 1:n
    x = x/2
  end
    x
end
```

```
f2 (generic function with 1 method)
```

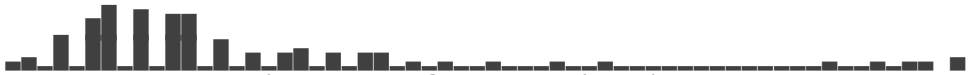
```
function f2(n)
  x=1.0
  for i = 1:n
    x = x/2.0
  end
x
end
```

```
BenchmarkTools.Trial: 10000 samples with 999 evaluations.
 Range (min … max):  7.982 ns … 38.852 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     8.024 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   8.209 ns ±  1.823 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
Memory estimate: 0 bytes, allocs estimate: 0.
```

```
@benchmark f1(10)
```

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  1.209 ns … 31.105 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     1.215 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   1.235 ns ±  0.627 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```




```
Memory estimate: 0 bytes, allocs estimate: 0.
```

```
@benchmark f2(10)
```

- Observation: function `f2` is faster than `f1` for the same operations.

```
	.text
; ┌ @ nb05-julia-jit.jl#==#fb6974d6-01e3-11eb-258b-9db21b4c39dd:1 within `f1'
	pushq	%rax
; │ @ nb05-julia-jit.jl#==#fb6974d6-01e3-11eb-258b-9db21b4c39dd:3 within `f1'
; │┌ @ range.jl:5 within `Colon'
; ││┌ @ range.jl:287 within `UnitRange'
; │││┌ @ range.jl:292 within `unitrange_last'
	testq	%rsi, %rsi
; │└└└
	jle	L123
; │ @ nb05-julia-jit.jl#==#fb6974d6-01e3-11eb-258b-9db21b4c39dd within `f1'
	movq	%rsi, %rax
	sarq	$63, %rax
	andnq	%rsi, %rax, %rax
; │ @ nb05-julia-jit.jl#==#fb6974d6-01e3-11eb-258b-9db21b4c39dd:4 within `f1'
	decq	%rax
	movb	$1, %sil
	movl	$1, %ecx
	xorl	%r8d, %r8d
```

```
with_terminal() do
        @code_native f1(10)
end
```

```
    .text
; ┌ @ nb05-julia-jit.jl#==#36244b3c-01e4-11eb-3828-2fa69b8b0835:1 within `f2'
    movabsq $.rodata.cst8, %rcx
; │ @ nb05-julia-jit.jl#==#36244b3c-01e4-11eb-3828-2fa69b8b0835:3 within `f2'
; │┌ @ range.jl:5 within `Colon'
; ││┌ @ range.jl:287 within `UnitRange'
; │││┌ @ range.jl:292 within `unitrange_last'
    testq   %rdi, %rdi
; │└└└
    jle L58
; │ @ nb05-julia-jit.jl#==#36244b3c-01e4-11eb-3828-2fa69b8b0835 within `f2'
    movq    %rdi, %rax
    sarq    $63, %rax
    andnq   %rdi, %rax, %rax
    vmovsd  (%rcx), %xmm0                   # xmm0 = mem[0],zero
    movabsq $140061776933888, %rcx          # imm = 0x7F62AC75F400
    vmovsd  (%rcx), %xmm1                   # xmm1 = mem[0],zero
    nopl    (%rax)
; │ @ nb05-julia-jit.jl#==#36244b3c-01e4-11eb-3828-2fa69b8b0835:4 within `f2'
```

```
with_terminal() do
        @code_native f2(10)
end
```

```
Variables
  #self#::Core.Const(Main.workspace2.f1)
  n::Int64
  @_3::Union{Nothing, Tuple{Int64, Int64}}
  x::Union{Float64, Int64}
  i::Int64

Body::Union{Float64, Int64}
1 ─       (x = 1)
│   %2  = (1:n)::Core.PartialStruct(UnitRange{Int64}, Any[Core.Const(1), Int64]
│         (@_3 = Base.iterate(%2))
│   %4  = (@_3 === nothing)::Bool
│   %5  = Base.not_int(%4)::Bool
└──       goto #4 if not %5
2 ┄ %7  = @_3::Tuple{Int64, Int64}::Tuple{Int64, Int64}
│         (i = Core.getfield(%7, 1))
│   %9  = Core.getfield(%7, 2)::Int64
```

```
with_terminal() do
        @code_warntype f1(10)
end
```

```
Variables
  #self#::Core.Const(Main.workspace2.f2)
  n::Int64
  @_3::Union{Nothing, Tuple{Int64, Int64}}
  x::Float64
  i::Int64

Body::Float64
1 ─       (x = 1.0)
│   %2  = (1:n)::Core.PartialStruct(UnitRange{Int64}, Any[Core.Const(1), Int64]
│         (@_3 = Base.iterate(%2))
│   %4  = (@_3 === nothing)::Bool
│   %5  = Base.not_int(%4)::Bool
└──       goto #4 if not %5
2 ┄ %7  = @_3::Tuple{Int64, Int64}::Tuple{Int64, Int64}
│         (i = Core.getfield(%7, 1))
│   %9  = Core.getfield(%7, 2)::Int64
```

```
with_terminal() do
        @code_warntype f2(10)
end
```

- Once again, "boxing" occurs to handle x: in `g()` it changes its type from Int64 to Float64. We see this with the union type for x in @code_warntype
- Conclusion: **Avoid Julia Gotcha #2 by ensuring variables keep their type also in functions**.

Gotcha #6: allocations

```
mymat =
10×100000 Matrix{Float64}:
 0.953273     0.854619  0.217856   0.522739  …  0.484237  0.552183   0.494694
 0.9426       0.919631  0.967124   0.707857     0.412788  0.335066   0.452799
 0.350204     0.95215   0.260878   0.74824      0.937667  0.406265   0.985637
 0.218863     0.581158  0.280101   0.855488     0.839223  0.40491    0.107036
 0.144156     0.890347  0.701138   0.535328     0.556455  0.0256335  0.420654
 0.662491     0.288674  0.812625   0.1458    …  0.18872   0.387899   0.22315
 0.0862631    0.131252  0.549655   0.826499     0.314158  0.61167    0.749936
 0.000982333  0.858958  0.656768   0.904339     0.702393  0.862879   0.0221123
 0.103138     0.440022  0.3642     0.394922     0.735594  0.514835   0.771918
 0.387846     0.668235  0.0914743  0.243121     0.692356  0.858902   0.639465
```

```
mymat=rand(10,100_000)
```

- Define three different ways of summing of squares of matrix columns:

```
g1 (generic function with 1 method)
```

```
function g1(a)
    y=0.0
    for j=1:size(a,2)
        for i=1:size(a,1)
            y=y+a[i,j]^2
        end
    end
    y
end
```

```
g2 (generic function with 1 method)
```

```
function g2(a)
    y=0.0
    for j=1:size(a,2)
        y=y+mapreduce(z->z^2,+,a[:,j])
    end
    y
end
```

```
g3 (generic function with 1 method)
```

```
function g3(a)
    y=0.0
    for j=1:size(a,2)
        @views y=y+mapreduce(z->z^2,+,a[:,j])
    end
    y
end
```

```
true
```

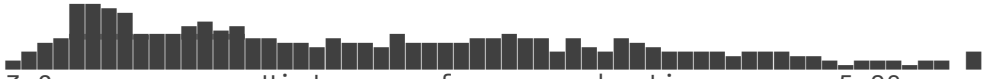
```
g1(mymat)≈ g2(mymat) && g2(mymat)≈ g3(mymat)
```

```
BenchmarkTools.Trial: 5208 samples with 1 evaluation.
 Range (min … max):  915.552 μs …  1.901 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     939.963 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   956.326 μs ± 50.176 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
 Memory estimate: 16 bytes, allocs estimate: 1.
```

```
@benchmark g1(mymat)
```

```
BenchmarkTools.Trial: 1297 samples with 1 evaluation.
 Range (min … max):  3.202 ms …   9.153 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     3.709 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   3.849 ms ± 503.977 μs  ┊ GC (mean ± σ):  6.72% ± 8.25%
```




```
 Memory estimate: 15.26 MiB, allocs estimate: 100001.
```

```
@benchmark g2(mymat)
```

```
BenchmarkTools.Trial: 6045 samples with 1 evaluation.
 Range (min … max):  711.479 μs …   2.025 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     773.489 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   822.752 μs ± 125.610 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 711 μs        Histogram: log(frequency) by time       1.33 ms <

Memory estimate: 16 bytes, allocs estimate: 1.
```

```
@benchmark g3(mymat)
```

- Observation: g3 is the fastest implemetation, then comes g1 and then g2.
- The difference between g2 and g1 is that each time we use a matrix slice `a[:,i]`, memory is allocated and data copied. Only then the mapreduce is employed, and the intermediate memory is garbage collected.
- The difference between g2 and g1 lies in the use of the `@views` macro which allows to avoid the creation of intermediae memory for matrix rows.
- Conclusion: avoid **Gotcha #6** **by carefully checking your code for allocations** and avoiding the use of temporary memory.