Advanced Development

How to not write bad code

Write more than just code!

- Comments!
- Tests!
- Documentation!

The rest of this session just concerns the code itself.

Preparations

Of course we load OSCAR

In [1]:
```
using Oscar
```

```
 ___   ___   ___    _    ____
 / _ \ / __\ / __\  / \  |  _ \  | Combining and extending ANTIC, GAP,
| |_| |\__ \| |__  / ^ \ |  ´ /  | Polymake and Singular
 \___/ \___/ \___//_/ \_\|_|\_\  | Type "?Oscar" for more information
o--------o-----o-----o--------o  | Documentation: https://docs.oscar-system.
org
  S Y M B O L I C   T O O L S    | Version 1.5.0
```

We install some Packages into the global environment to use them later in this talk: `Chairmarks`, `Profile`, `PProf`, `ProfileView`

Your turn: Install these packages now into you default environment.

Two fundamental principles:

1. Ensure that the compiler can infer the type of every variable.
2. Avoid unnecessary heap allocations.

All performance tips come down to these two principles.

The compiler's job is to optimize and translate Julia code it into runnable machine code. If a variable's type cannot be deduced before the code is run, then the compiler won't generate efficient code to handle that variable. We call this phenomenon "type

instability". Enabling type inference means making sure that every variable's type in every function can be deduced from the types of the function inputs alone.

A "heap allocation" (or simply "allocation") occurs when we create a new variable without knowing how much space it will require (like a Vector with flexible length). Julia has a mark-and-sweep garbage collector (GC), which runs periodically during code execution to free up space on the heap. Execution of code is stopped while the garbage collector runs, so minimising its usage is important.

Benchmarking code

How to measure how fast some code is

In [2]:
```
F, (x,y) = free_associative_algebra(QQ, [:x, :y])
A = matrix(F, [rand(F, 0:5, -10:10, -10:10) for i in 1:3, j in 1:3])
```

Out[2]:
```
[                           0   -7//8       7]
[-4//5*x*y + 1//2*y^2 + 17//8   -2//5       x]
[                         4//3       0   -4//7]
```

In [3]:
```
@time A^2; # Inaccurate, note the >90% compilation time
```

```
 0.345210 seconds (260.90 k allocations: 12.853 MiB, 99.98% compilation tim
e)
```

In [4]:
```
@time A^2
```

```
 0.000044 seconds (311 allocations: 14.922 KiB)
```

Out[4]:
```
[     7//10*x*y - 7//16*y^2 + 1435//192                                  7//2
0                                    -7//8*x - 4]
[8//25*x*y - 1//5*y^2 + 4//3*x - 17//20   7//10*x*y - 7//16*y^2 - 2719//160
0   -28//5*x*y + 7//2*y^2 - 34//35*x + 119//8]
[                                  -16//21                                -7//
6                                      1420//147]
```

`@time` only runs code a single time, and is thus heavily influenced by compilation, other things running on your machine, stochastical outliers etc.

Your turn: Run `@time sieve(1000)`.

Better: run code multiple times and take the minimum

In [5]:
```
using Chairmarks
@b A^2
```

Out[5]:
```
 10.433 µs (311 allocs: 14.922 KiB)
```

In [6]:
```
@be A^2
```

Out[6]:
```
Benchmark: 2979 samples with 3 evaluations
 min    7.453 μs (311 allocs: 14.922 KiB)
 median 8.901 μs (311 allocs: 14.922 KiB)
 mean   10.184 μs (311 allocs: 14.922 KiB)
 max    259.322 μs (311 allocs: 14.922 KiB)
```

Your turn: Run `@b sieve(1000)`.

There are many options including `init`, `setup`, `teardown` and keyword arguments, see the Chairmarks documentation for details.

Profiling code

Where is most of the time spent

A sampling profiler asks the program periodically for the currently executed line.

In julia: the package `Profile` offers a runtime profiler in `Profile` and a memory profiler in `Profile.Allocs`. Visualization is possible using `PProf.jl` or `ProfileView.jl`.

In [45]:
```
using Profile
Profile.clear()
Profile.Allocs.clear()
```

In [46]:
```
using Profile, PProf
A = matrix(F, [rand(F, 0:5, -10:10, -10:10) for i in 1:25, j in 1:25])
Profile.@profile A^2;
pprof()
```

Out[46]:
```
"profile.pb.gz"
```

```
Main binary filename not available.
Serving web UI on http://localhost:57599
```

In [47]:
```
Profile.Allocs.@profile A^2;
PProf.Allocs.pprof()
```

```
Analyzing 35858 allocation samples... 100%|████████████| Time: 0:00:18
```

Out[47]:
```
"alloc-profile.pb.gz"
```

```
Main binary filename not available.
Serving web UI on http://localhost:62261
pprof: listen tcp 127.0.0.1:62261: bind: address already in use
```

Your turn: Run `Profile.Allocs.@profile sieve(1000)`. Afterwards run `PProf.Allocs.pprof()`.

Type stability

Goal: the compiler should be able to infer "concrete" types for each variable

In [10]:
```
isconcretetype(Any), isconcretetype(Vector)
```

Out[10]:
```
(false, false)
```

```
Main binary filename not available.
Serving web UI on http://localhost:62261
pprof: listen tcp 127.0.0.1:62261: bind: address already in use
```

In [11]:
```
isconcretetype(ZZRingElem), isconcretetype(Vector{ZZRingElem})
```

Out[11]:
```
(true, true)
```

In particular: Types of variables and returned objects should **only depend on the type** of the input and **not on the value**.

If all variables are concretely inferred, a function call is just a `GOTO` statement. Otherwise, the code first has to list all matching methods and decide which of these is most specific ("dynamic dispatch"). This prevents further optimizations.

Example 1

In [12]:
```
function clamp(x)
    if x < 0
        return 0
    elseif x > 1
        return 1
    else
        return x
    end
end
```

Out[12]:
```
clamp (generic function with 1 method)
```

In [13]:
```
@code_warntype clamp(QQ(1,3))
```

```
MethodInstance for clamp(::QQFieldElem)
  from clamp(x) @ Main In[12]:1
Arguments
  #self#::Core.Const(Main.clamp)
  x::QQFieldElem
Body::Union{Int64, QQFieldElem}
1 ─ %1 = Main.:<::Core.Const(<)
│   %2 = (%1)(x, 0)::Bool
└──      goto #3 if not %2
2 ─      return 0
3 ─ %5 = Main.:>::Core.Const(>)
│   %6 = (%5)(x, 1)::Bool
└──      goto #5 if not %6
4 ─      return 1
5 ─ %9 = x::QQFieldElem
└──      return %9
```

In [14]:
```
function clamp2(x::T) where T
    if x < 0
        return T(0)
    elseif x > 1
        return T(1)
    else
        return x
    end
end
```

Out[14]:
```
clamp2 (generic function with 1 method)
```

In [15]:
```
@code_warntype clamp2(QQ(1,3))
```

```
MethodInstance for clamp2(::QQFieldElem)
  from clamp2(x::T) where T @ Main In[14]:1
Static Parameters
  T = QQFieldElem
Arguments
  #self#::Core.Const(Main.clamp2)
  x::QQFieldElem
Body::QQFieldElem
1 ─ %1  = (x < 0)::Bool
└──       goto #3 if not %1
2 ─ %3  = $(Expr(:static_parameter, 1))::Core.Const(QQFieldElem)
│   %4  = (%3)(0)::QQFieldElem
└──       return %4
3 ─ %6  = (x > 1)::Bool
└──       goto #5 if not %6
4 ─ %8  = $(Expr(:static_parameter, 1))::Core.Const(QQFieldElem)
│   %9  = (%8)(1)::QQFieldElem
└──       return %9
5 ─ %11 = x::QQFieldElem
└──       return %11
```

Learning: The return type should not depend on the input value, just on the input type.

Example 2

In [16]:
```
function put_in_vec_and_sum(x)
    v = []
    push!(v, x)
    return sum(v)
end
put_in_vec_and_sum(ZZ(42))
```

Out[16]: 42

In [17]:
```
@code_warntype put_in_vec_and_sum(ZZ(42))
```

```
MethodInstance for put_in_vec_and_sum(::ZZRingElem)
  from put_in_vec_and_sum(x) @ Main In[16]:1
Arguments
  #self#::Core.Const(Main.put_in_vec_and_sum)
  x::ZZRingElem
Locals
  v::Vector{Any}
Body::Any
1 ─      (v = Base.vect())
│   %2 = Main.push!::Core.Const(push!)
│   %3 = v::Vector{Any}
│        (%2)(%3, x)
│   %5 = Main.sum::Core.Const(sum)
│   %6 = v::Vector{Any}
│   %7 = (%5)(%6)::Any
└──      return %7
```

In [18]:
```
function put_in_vec_and_sum2(x::T) where T
    v = T[]
    push!(v, x)
    return sum(v)
end
put_in_vec_and_sum2(ZZ(42))
```

Out[18]: 42

In [19]:
```
@code_warntype put_in_vec_and_sum2(ZZ(42))
```

```
MethodInstance for put_in_vec_and_sum2(::ZZRingElem)
  from put_in_vec_and_sum2(x::T) where T @ Main In[18]:1
Static Parameters
  T = ZZRingElem
Arguments
  #self#::Core.Const(Main.put_in_vec_and_sum2)
  x::ZZRingElem
Locals
  v::Vector{ZZRingElem}
Body::ZZRingElem
1 ─ %1 = $(Expr(:static_parameter, 1))::Core.Const(ZZRingElem)
│        (v = Base.getindex(%1))
│   %3 = v::Vector{ZZRingElem}
│        Main.push!(%3, x)
│   %5 = v::Vector{ZZRingElem}
│   %6 = Main.sum(%5)::ZZRingElem
└──      return %6
```

Learning: Don't use abstractly typed containers (unless you have a good reason to do so).

Example 3

In [20]:
```
function reciprocal(x::Union{ZZRingElem, QQFieldElem})
    x = QQ(x)
    x = 1//x
    return x
end
```

Out[20]: reciprocal (generic function with 1 method)

In [21]:
```
@code_warntype reciprocal(ZZ(5))
```

```
MethodInstance for reciprocal(::ZZRingElem)
  from reciprocal(x::Union{QQFieldElem, ZZRingElem}) @ Main In[20]:1
Arguments
  #self#::Core.Const(Main.reciprocal)
  x@_2::ZZRingElem
Locals
  x@_3::Union{QQFieldElem, ZZRingElem}
Body::QQFieldElem
1 ─      (x@_3 = x@_2)
│   %2 = x@_3::ZZRingElem
│        (x@_3 = Main.QQ(%2))
│   %4 = Main.://::Core.Const(//)
│   %5 = x@_3::QQFieldElem
│        (x@_3 = (%4)(1, %5))
│   %7 = x@_3::QQFieldElem
└──      return %7
```

In [22]:
```
function reciprocal2(x_::Union{ZZRingElem, QQFieldElem})
    x = QQ(x_)
    x = 1//x
```

```
    return x
end
```

Out[22]: reciprocal2 (generic function with 1 method)

In [23]:
```
@code_warntype reciprocal2(ZZ(5))
```

```
MethodInstance for reciprocal2(::ZZRingElem)
  from reciprocal2(x_::Union{QQFieldElem, ZZRingElem}) @ Main In[22]:1
Arguments
  #self#::Core.Const(Main.reciprocal2)
  x_::ZZRingElem
Locals
  x::QQFieldElem
Body::QQFieldElem
1 ─      (x = Main.QQ(x_))
│   %2 = x::QQFieldElem
│        (x = 1 // %2)
│   %4 = x::QQFieldElem
└──      return %4
```

Learning: A local variable should only hold objects of a single type.

More OSCAR-idiomatic: Don't even allow `ZZRingElem` as input. All arithmetic functions should stay in the input ring.

How to use unstable functions in your code

In [24]:
```
function unstable_func(x)
    v = []
    push!(v, 2x^2 - 1)
    return sum(v)
end
```

Out[24]: unstable_func (generic function with 1 method)

In [25]:
```
function my_func(a,b,c,d)
    @req allequal(parent, (a,b,c,d)) "parent mismatch"
    m = matrix(parent(a), 2, 2, [a,b,c,d])
    r = unstable_func(m) # this function is unstable
    return det(r)
end
```

Out[25]: my_func (generic function with 1 method)

In [26]:
```
@code_warntype my_func(ZZ(1), ZZ(2), ZZ(3), ZZ(4))
```

```
MethodInstance for my_func(::ZZRingElem, ::ZZRingElem, ::ZZRingElem, ::ZZRin
gElem)
  from my_func(a, b, c, d) @ Main In[25]:1
Arguments
  #self#::Core.Const(Main.my_func)
  a::ZZRingElem
  b::ZZRingElem
  c::ZZRingElem
  d::ZZRingElem
Locals
  r::Any
  m::ZZMatrix
Body::Any
1 ─       Core.NewvarNode(:(r))
│         Core.NewvarNode(:(m))
│   %3  = AbstractAlgebra.:!::Core.Const(!)
│   %4  = Main.allequal::Core.Const(allequal)
│   %5  = Main.parent::Core.Const(parent)
│   %6  = Core.tuple(a, b, c, d)::NTuple{4, ZZRingElem}
│   %7  = (%4)(%5, %6)::Core.Const(true)
│   %8  = (%3)(%7)::Core.Const(false)
└──       goto #3 if not %8
2 ─       Core.Const(:(AbstractAlgebra.throw))
│         Core.Const(:(AbstractAlgebra.ArgumentError("parent mismatch")))
└──       Core.Const(:((%10)(%11)))
3 ┄ %13 = Main.matrix::Core.Const(AbstractAlgebra.matrix)
│   %14 = Main.parent::Core.Const(parent)
│   %15 = (%14)(a)::Core.Const(Integer ring)
│   %16 = Base.vect(a, b, c, d)::Vector{ZZRingElem}
│         (m = (%13)(%15, 2, 2, %16))
│   %18 = m::ZZMatrix
│         (r = Main.unstable_func(%18))
│   %20 = Main.det::Core.Const(LinearAlgebra.det)
│   %21 = r::Any
│   %22 = (%20)(%21)::Any
└──       return %22
```

In [27]:
```
function my_func2(a,b,c,d)
    @req allequal(parent, (a,b,c,d)) "parent mismatch"
    m = matrix(parent(a), 2, 2, [a,b,c,d])
    r = unstable_func(m)::typeof(m)
    return det(r)
end
```

Out[27]:
```
my_func2 (generic function with 1 method)
```

In [28]:
```
@code_warntype my_func2(ZZ(1), ZZ(2), ZZ(3), ZZ(4))
```

```
MethodInstance for my_func2(::ZZRingElem, ::ZZRingElem, ::ZZRingElem, ::ZZRi
ngElem)
  from my_func2(a, b, c, d) @ Main In[27]:1
Arguments
  #self#::Core.Const(Main.my_func2)
  a::ZZRingElem
  b::ZZRingElem
  c::ZZRingElem
  d::ZZRingElem
Locals
  r::ZZMatrix
  m::ZZMatrix
Body::ZZRingElem
1 ─       Core.NewvarNode(:(r))
│         Core.NewvarNode(:(m))
│   %3  = AbstractAlgebra.:!::Core.Const(!)
│   %4  = Main.allequal::Core.Const(allequal)
│   %5  = Main.parent::Core.Const(parent)
│   %6  = Core.tuple(a, b, c, d)::NTuple{4, ZZRingElem}
│   %7  = (%4)(%5, %6)::Core.Const(true)
│   %8  = (%3)(%7)::Core.Const(false)
└──       goto #3 if not %8
2 ─       Core.Const(:(AbstractAlgebra.throw))
│         Core.Const(:(AbstractAlgebra.ArgumentError("parent mismatch")))
└──       Core.Const(:((%10)(%11)))
3 ┄ %13 = Main.matrix::Core.Const(AbstractAlgebra.matrix)
│   %14 = Main.parent(a)::Core.Const(Integer ring)
│   %15 = Base.vect(a, b, c, d)::Vector{ZZRingElem}
│         (m = (%13)(%14, 2, 2, %15))
│   %17 = m::ZZMatrix
│   %18 = Main.unstable_func(%17)::Any
│   %19 = Main.typeof::Core.Const(typeof)
│   %20 = m::ZZMatrix
│   %21 = (%19)(%20)::Core.Const(ZZMatrix)
│         (r = Core.typeassert(%18, %21))
│   %23 = r::ZZMatrix
│   %24 = Main.det(%23)::ZZRingElem
└──       return %24
```

In [29]:
```
@b my_func(ZZ(1), ZZ(2), ZZ(3), ZZ(4))
```

Out[29]:
```
  691.000 ns (17 allocs: 656 bytes)
```

In [30]:
```
@b my_func2(ZZ(1), ZZ(2), ZZ(3), ZZ(4))
```

Out[30]:
```
  666.375 ns (17 allocs: 656 bytes)
```

Allocations

We can measure allocations using `@time` or `@b`.

In [31]:
```
F, (x,y) = free_associative_algebra(QQ, [:x, :y])
A = matrix(F, [rand(F, 0:5, -10:10, -10:10) for i in 1:3, j in 1:3])
```

```
@b A^2
```

```
Out[31]:  7.865 µs (294 allocs: 17.859 KiB)
```

Many allocations are often (but not necessarilty) a sign of sub-optimal code.

Example 1

Creating a polynomial ring

```
In [32]: R = QQ; n = 5;
```

```
In [33]: @b S, _ = polynomial_ring(R, "x" => 1:n, "y" => 1:n)
```

```
Out[33]:  7.005 µs (173 allocs: 6.359 KiB)
```

```
In [34]: @b S, _ = polynomial_ring(QQ, :x => 1:n, :y => 1:n)
```

```
Out[34]:  5.692 µs (109 allocs: 4.609 KiB)
```

Learning: Think twice if you really need strings.

Example 2

Getting generators of a polynomial ring

```
In [35]: S, _ = polynomial_ring(QQ, :x => 1:n, :y => 1:n)
         # [...]
         @b gens(S)[1:n]
```

```
Out[35]:  1.503 µs (35 allocs: 1.516 KiB)
```

```
In [36]: S, x, y = polynomial_ring(QQ, :x => 1:n, :y => 1:n)
         x
```

```
Out[36]: 5-element Vector{QQMPolyRingElem}:
          x[1]
          x[2]
          x[3]
          x[4]
          x[5]
```

Learning: Don't throw away things just to recompute them again.

Example 3

Getting the number of generators of a polynomial ring

In [37]:
```
@b length(gens(S))
```

Out[37]:
```
1.040 µs (32 allocs: 1.391 KiB)
```

In [38]:
```
@b ngens(S)
```

Out[38]:
```
14.446 ns
```

Learning: Use dedicated functions, if available.

Example 4

Task: given a square matrix `A` and a positive integer `n`, compute `det(A^n)`

In [39]:
```
A = matrix(ZZ, [-500 0 43 2; 33 12 0 2; -1 9 -55 2^20; 1 5 -1 -1]); n = 50;
```

In [40]:
```
function task(A::MatrixElem, n::Int)
    @req is_square(A) "matrix is not square"
    return det(A^n)
end
@b task(A, n)
```

Out[40]:
```
12.731 µs (10.50 allocs: 1.305 KiB)
```

In [41]:
```
function task2(A::MatrixElem, n::Int)
    @req is_square(A) "matrix is not square"
    return det(A)^n
end
@b task2(A, n)
```

Out[41]:
```
459.918 ns (2.94 allocs: 188.735 bytes)
```

Learning: Use mathematics!

Most important:

If you think that some dedicated function should be there but you don't find it, ask!

If you think that something should be faster, ask!
(not just this week, but also afterwards. slack, github, mail, etc.)

Code must not be perfect!
(It just needs to be good enough to help with your research)

Some further reading:

- Chairmarks.jl documentation for more benchmarking options

- Julia manual on performance tips
- JET.jl and Cthulhu.jl for examining type stability deeper in the call stack

Your turn: Optimize your `sieve` function. `@code_warntype sieve(1000)` and the allocation profiling you already did should be some good starting points. Compare your intermediate versions to the initial one using `@b sieve(1000)`.