Selected Features of OSCAR

Daniel Schultz

TU Kaiserslautern

ICERM Feb 16, 2021

OSCAR (open source computer algebra resource) comprises:

- Singular.jl: algebraic geometry
- GAP.jl: group theory
- Polymake.jl: polyhedral geometry
- Hecke.jl: algebraic number theory
- Nemo.jl: specific implementations of common rings
- AbstractAlgebra.jl: type system for mathematical objects and generic implementations of many rings and operations

julia:

- elaborate type system
- dynamic language
- JIT compilation
- multiple dispatch

```
julia> M = MatrixAlgebra(ResidueField(ZZ, 5), 2)
Matrix Algebra of degree 2 over Residue field of Integers modulo 5
```

```
julia> R, x = PolynomialRing(M, "x")
(Univariate Polynomial Ring in x over Matrix Algebra of degree 2
 over Residue field of Integers modulo 5, x)
```

```
julia> typeof(R)
AbstractAlgebra.Generic.NCPolyRing{
    AbstractAlgebra.Generic.MatAlgElem{
        AbstractAlgebra.Generic.ResF{BigInt}}}
```

```
julia> typeof(x)
AbstractAlgebra.Generic.NCPoly{
    AbstractAlgebra.Generic.MatAlgElem{
        AbstractAlgebra.Generic.ResF{BigInt}}}
```

```
julia> divexact_left(M([1 2; 3 4])*x^2, M([4 1; 2 1])*x)
[4 4; 0 1]*x
```

```
julia> divexact_right(M([1 2; 3 4])*x^2, M([4 1; 2 1])*x)
[1 1; 0 4]*x
```

Nemo.jl: $\mathbb{C}$, $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{F}_q$, $\mathbb{Q}_p$, and $\mathbb{Q}_q$, and polynomials, power series, and matrices.

- fast multiplication of polynomials and matrices
- generic algorithms for polynomial gcd are generically terrible
- generic algorithms for polynomial factorization are nonexistent

```
julia> Qxyz, (x, y, z) = PolynomialRing(QQ, ["x", "y", "z"])
(Multivariate Polynomial Ring in x, y, z over Rational Field,
 fmpq_mpoly[x, y, z])
```

```
julia> M = matrix(Qxyz, [x y z; x^2 y^2 z^2; x^3 y^3 z^3]);
       @less det(M)
function det(M::MatrixElem{T}) where {T <: RingElement}
   !issquare(M) && error("Not a square matrix in det")
...
```

```
julia> @less x + y
function +(a::fmpq_mpoly, b::fmpq_mpoly)
   check_parent(a, b)
   z = parent(a)()
   ccall((:fmpq_mpoly_add, libflint), Nothing,
         (Ref{fmpq_mpoly}, Ref{fmpq_mpoly},
          Ref{fmpq_mpoly}, Ref{FmpqMPolyRing}),
         z, a, b, a.parent)
   return z
end
```

The (private) internals of elements of $\mathbb{F}_p[x, y, z, t]$

```
julia> _, (x, y, z, t) = PolynomialRing(GF(5),["x", "y", "z", "t"])
(Multivariate Polynomial Ring in x, y, z, t over Galois field with
 characteristic 5, gfp_mpoly[x, y, z, t])
```

The polynomials are stored internally in fully expanded form as a sum of products of a coefficient and a monomial. The actual type definition:

```
mutable struct gfp_mpoly <: MPolyElem{gfp_elem}
   coeffs::Ptr{Nothing}
   exps::Ptr{Nothing}
   length::Int
   bits::Int
   coeffs_alloc::Int
   exps_alloc::Int
```

Function for viewing the polynomial internals

```
function view(a::Ptr{UInt}, n, leftpad)
    for i in 1:n
        print(" "^(i == 1 ? 0 : leftpad))
        println(string(unsafe_load(a, i), base = 16,
                                          pad = 2*sizeof(UInt)))
    end
end

function view(poly::gfp_mpoly)
    @show poly
    println("length: ", poly.length)
    print("coeffs: ")
    view(reinterpret(Ptr{UInt}, poly.coeffs), poly.coeffs_alloc, 8)
    println("  bits: ", poly.bits)
    print("  exps: ")
    view(reinterpret(Ptr{UInt}, poly.exps), poly.exps_alloc, 8)
end
```

```
julia> view(3*x*y*z*t + x + 2y^2 + 3z^3 + 4t^4 + x*y*z*t)
poly = 4*x*y*z*t + x + 2*y^2 + 3*z^3 + 4*t^4
length: 5
coeffs: 0000000000000004
        0000000000000001
        0000000000000002
        0000000000000003
        0000000000000004
        0000000002c61000
  bits: 16
  exps: 0001000100010001
        0001000000000000
        0000000200000000
        0000000000030000
        0000000000000004
        0000000002c61000
```

```
julia> p = x^Int(0xFFFFFFFFFFF) + y; view(p)
poly = x^17592186044415 + y
length: 2
coeffs: 0000000000000001
        0000000000000001
  bits: 64
  exps: 0000000000000000
        0000000000000000
        0000000000000000
        00000fffffffffff
        0000000000000000
        0000000000000000
        0000000000000001
        0000000000000000
```

Do not do this!

```
julia> unsafe_store!(reinterpret(Ptr{UInt}, p.exps), 10, 6); p
x^17592186044415 + y*z^10
```

safer but still unsafe:

```
julia> set_exponent_vector!(p, 1, [0,1,2,3]); sort_terms!(p)
y*z^10 + y*z^2*t^3
```

The polynomials are not black boxes: we have

- iterators for the coefficients
- iterators for the exponent vectors
- (they work across all polynomial implementations)

You can also construct a polynomial from:

- basic arithmetic operations on constants and variables (gens)
- coefficients and exponent vectors
- (these work across all polynomial implementations)

Let's see these in action to swap the variables in a polynomial.

```
julia> _, (x, y) = ZZ["x", "y"]; (2*x^2+3*y)(y, x)
3*x + 2*y^2
```

First try:

```
julia> function swap_vars(p)
           x, y = gens(parent(p))
           result = zero(parent(p))
           for (c, exps) in zip(coeffs(p), exponent_vectors(p))
               result += c*y^exps[1]*x^exps[2]
           end
           return result
       end
julia> swap_vars(2*x^2 + 3*y)
3*x + 2*y^2
```

```
julia> for n in 10:15
           p = divexact(x^2^n - y^2^n, x - y)
           @time swap_vars(p)
       end
  0.002201 seconds (19.46 k allocations: 8.711 MiB)
  0.018249 seconds (38.92 k allocations: 33.422 MiB, 14.85% gc time)
  0.045636 seconds (77.83 k allocations: 130.844 MiB, 26.28% gc time)
  0.189864 seconds (155.65 k allocations: 517.688 MiB, 41.69% gc time)
  0.447574 seconds (311.30 k allocations: 2.011 GiB, 2.80% gc time)
  1.788859 seconds (622.60 k allocations: 8.022 GiB, 2.37% gc time)
```

It’s quadratic!

Second try:

```
julia> function swap_vars(p)
           result = MPolyBuildCtx(parent(p))
           for (c, exps) in zip(coeffs(p), exponent_vectors(p))
               push_term!(result, c, [exps[2], exps[1]])
           end
           return finish(result)
       end
julia> swap_vars(2*x^2 + 3*y)
3*x + 2*y^2
```

```
julia> for n in 10:15
           p = divexact(x^2^n - y^2^n, x - y)
           @time swap_vars(p)
       end
  0.000119 seconds (3.10 k allocations: 224.461 KiB)
  0.000231 seconds (6.18 k allocations: 448.461 KiB)
  0.000484 seconds (12.32 k allocations: 896.461 KiB)
  0.000986 seconds (24.61 k allocations: 1.750 MiB)
  0.002086 seconds (49.19 k allocations: 3.500 MiB)
  0.004085 seconds (98.35 k allocations: 7.000 MiB)
```

It’s linear!

```
julia> _, (x, y) = Singular.PolynomialRing(Singular.ZZ, ["x", "y"])
(Singular Polynomial Ring (ZZ),(x,y),(dp(2),C), spoly{n_Z}[x, y])
```

```
julia> swap_vars(2*x^2 + 3*y)   # now compiled for spoly{n_Z}
2*y^2 + 3*x
```

compiler already knew this:

```
julia> typeof(ans)        # @code_warntype swap_vars(2*x^2 + 3*y)
spoly{n_Z}
```

```
julia> for n in 10:15
           p = divexact(x^2^n - y^2^n, x - y)
           @time swap_vars(p)
       end
  0.002593 seconds (19.46 k allocations: 472.141 KiB)
  0.001839 seconds (43.01 k allocations: 1008.031 KiB)
  0.003160 seconds (86.02 k allocations: 1.969 MiB)
  0.005271 seconds (172.03 k allocations: 3.938 MiB)
  0.010631 seconds (344.06 k allocations: 7.875 MiB)
  0.022158 seconds (688.13 k allocations: 15.750 MiB)
```

Third try (in-place operation):

```
julia> function swap_vars!(p)
           exps = zeros(Int, nvars(parent(p)))
           for i in 1:length(p)
               exps[1] = exponent(p, i, 2)
               exps[2] = exponent(p, i, 1)
               set_exponent_vector!(p, i, exps)
           end
           return sort_terms!(p)
       end
julia> a = b = 2*x^2 + 3*y; swap_vars!(a); (a, b)
(3*x + 2*y^2, 3*x + 2*y^2)
```

```
julia> for n in 10:15
           p = divexact(x^2^n - y^2^n, x - y)
           @time swap_vars!(p)
       end
  0.000072 seconds (1 allocation: 96 bytes)
  0.000136 seconds (1 allocation: 96 bytes)
  0.000280 seconds (1 allocation: 96 bytes)
  0.000559 seconds (1 allocation: 96 bytes)
  0.001199 seconds (1 allocation: 96 bytes)
  0.002280 seconds (1 allocation: 96 bytes)
```

It's in-place and the only allocation is in `exps =`

Creating your own fields in OSCAR

Suppose we want to work in the fraction field of a GCD domain while avoiding unnecessary expansions.

The result of $\frac{1}{x+y} + \frac{1}{x+y+1}$ should look like and *be* $\frac{2x+2y+1}{(x+y)(x+y+1)}$.

```
julia> _, (x, y) = ZZ["x", "y"]; 1//(x + y) + 1//(x + y + 1)
(2*x + 2*y + 1)//(x^2 + 2*x*y + x + y^2 + y)

julia> typeof(ans)
AbstractAlgebra.Generic.Frac{AbstractAlgebra.Generic.MPoly{BigInt}}
```

Definition of `AbstractAlgebra.Generic.Frac`:

```
mutable struct Frac{T <: RingElem} <: AbstractAlgebra.FracElem{T}
   num::T
   den::T
   parent::FracField{T}
```

Creating your own fields in OSCAR

For a generic GCD domain R, instead of elements of $\mathrm{frac}(R)$ being

$$\frac{a}{b} \quad a \in R, b \in R \setminus 0$$

they are going to be

$$u \prod_i b_i^{e_i}, e_i \in \mathbb{Z}$$

where $u \in R$ is either zero or a unit and the $b_i \in R$ are not zero.

The elements $u \prod_i b_i^{e_i}$ will be of type `FactoredElem{T}` where `T` is the type of the elements of R
The type of the parent object frac(R) will be of type `FactoredField{T}`

```
mutable struct FactoredField{T <: RingElement} <: Field
    base_ring::AbstractAlgebra.Ring
end

mutable struct FactoredElemTerm{T <: RingElement}
    base::T
    exp::Int
end

mutable struct FactoredElem{T <: RingElement} <: FieldElem
    unit::T
    terms::Vector{FactoredElemTerm{T}}
    parent::FactoredField{T}
end
```

Tell the system that the parent objects of type `FactoredField{T}` have elements of type `FactoredElem{T}`:

```
function elem_type(::Type{FactoredField{T}}) where {T <: RingElement}
    return FactoredElem{T}
end
```

External interface for constructing the new fraction field

```
function FactoredFractionField(R::AbstractAlgebra.Ring)
   return FactoredField{AbstractAlgebra.elem_type(R)}(R)
end
```

Printing of the parent object and elements

```
function Base.show(io::IO, F::FactoredField)
    print(io, "Factored fraction field of ", base_ring(F))
end

function Base.show(io::IO, a::FactoredElem)
    print(io, AbstractAlgebra.obj_to_string(a, context = io))
end

function expressify(a::FactoredElem; context = nothing)
    n = Expr(:call, :*, expressify(a.unit, context = context))
    d = Expr(:call, :*)
    for t in a.terms
        b = expressify(t.base; context = context)
        e = checked_abs(t.exp)
        push!((e == t.exp ? n : d).args,
              isone(e) ? b : Expr(:call, :^, b, e))
    end
    return length(d.args) < 2 ? n : Expr(:call, :/, n, d)
end
```

```
function +(a::FactoredElem{T}, b::FactoredElem{T}) where T
    (g, x, y) = gcdhelper(a, b)
    return g*(x + y)
end

function -(a::FactoredElem{T}, b::FactoredElem{T}) where T
    (g, x, y) = gcdhelper(a, b)
    return g*(x - y)
end

function ==(a::FactoredElem{T}, b::FactoredElem{T}) where T
    (g, x, y) = gcdhelper(a, b)
    return x == y
end

function *(a::FactoredElem{T}, b::T) where T <: RingElem
    F = parent(a)
    parent(b) == base_ring(F) || error("oops")
    iszero(b) && return zero(F)
    z = FactoredElem{T}(a.unit, map(copy, a.terms), F)
    append_pow_normalize!(z, b, 1, 1)
    return z
end
```

The is* functions

```
function isunit(a::FactoredElem{T}) where T
    return !iszero(a)
end

function iszero(a::FactoredElem{T}) where T
    return iszero(a.unit)
end

function isone(a::FactoredElem{T}) where T
    if iszero(a.unit)
        return false
    end
    for i in a.terms
        ok = !iszero(i.exp) && !isunit(i.base)
        for j in a.terms
            ok = ok && (j === i || isunit(gcd(i.base, j.base)))
        end
        ok && return false
    end
    z = normalize(a)
    return isempty(z.terms) && isone(z.unit)
end
```

The new field in action: $\mathbb{Q}$

```
julia> F = FactoredFractionField(ZZ)
Factored fraction field of Integers
```

```
julia> (F(4), F(6), F(4)*F(6))
(4, 6, 2^3*3)

julia> F(4)*F(6)-F(2//5)^2
2^2*149/5^2

julia> F(4)*F(6)-(2//5)^2
2^2*149/25

julia> sum(1/F(i) for i in 1:42)
12309312989335019/(2^5*3^3*17*41*5^2*23*13*19*37*31*29)
```

```
julia> @which F(4)*ZZ(6)
*(a::FactoredElem{T}, b::T) where T<:RingElem
 in Main at /home/schultz/FactoredElem.jl:344
```

The new field in action: $\mathbb{Q}(x, y, z)$

```
julia> F = FactoredFractionField(first(ZZ["x", "y", "z"]))
Factored fraction field of Multivariate Polynomial Ring in x, y, z
 over Integers
```

```
julia> (x, y, z) = map(F, gens(base_ring(F)))
3-element Vector{FactoredElem{AbstractAlgebra.Generic.MPoly{BigInt}}}:
 x
 y
 z
```

```
julia> 1/(x+y) + 1/(x+y+z)
(2*x + 2*y + z)/((x + y)*(x + y + z))
```

```
julia> det(matrix(F, [x y z; x^2 y^2 z^2; x^3 y^3 z^3]))
-x*(y - z)*(x - y)*y*(x - z)*z
```

Oscar.jl

... where $\mathbb{Q}[x, y]/(y^2 + x^3 + 1)$ works and much more!

```
julia> using Oscar
 -----    -----    -----      -      -----
|     |  |     |  |     |    | |    |     |
|     |  |        |         |   |   |     |
|     |   -----   |        |     |  |-----
|     |        |  |        |-----|  |   |
|     |  |     |  |     |  |     |  |    |
-----    -----    -----   -     -  -     -
...combining (and extending) ANTIC, GAP, Polymake and Singular
Version 0.5.1-DEV ...
... which comes with absolutely no warranty whatsoever
Type: ’?Oscar’ for more information
(c) 2019-2021 by The Oscar Development Team
```

Quotient Rings

```
julia> Qxy, (x, y) = PolynomialRing(QQ, ["x", "y"])
(Multivariate Polynomial Ring in x, y over Rational Field,
 fmpq_mpoly[x, y])
```

In $\mathbb{Q}[x, y]$ we do not have divisibility:

```
julia> divides(2*x^3 - x^2 - 3*x*y + 2, x + y)
(false, 0)
```

$R = \mathbb{Q}[x, y]/(x^3 + y^2 + 1)$:

```
julia> R, _ = quo(Qxy, y^2 + x^3 + 1); R
Quotient of Multivariate Polynomial Ring in x, y over Rational
 Field by ideal generated by: x^3 + y^2 + 1
```

In R we do:

```
julia> divides(R(2*x^3 - x^2 - 3*x*y + 2), R(x + y))
(true, -x - 2*y)
```

```
julia> typeof(ans[2])
Oscar.MPolyQuoElem{fmpq_mpoly}
```

Primary Decomposition

experimental

```
julia> Oscar.example("PrimDec.jl")

julia> R,(x,y,z) = PolynomialRing(QQ, ["x", "y", "z"]);
       i = ideal(R, [(z^2+1)*(z^3+2)^2, y-z^2])
ideal generated by: z^8 + z^6 + 4*z^5 + 4*z^3 + 4*z^2 + 4, y - z^2

julia> primary_decomposition(i)
4-element Array{Oscar.MPolyIdeal,1}:
 ideal generated by: z^2 + 1, y + 1
 ideal generated by: z^2 + 1, y + 1
 ideal generated by: z^6 + 4*z^3 + 4, y - z^2
 ideal generated by: z^3 + 2, y - z^2
```

Normalization

Compute the integral closure of $\mathbb{Q}[x, y, z]/(z - x^2, z - y^3)$ inside its field of fractions.

```
julia> R, (x, y, z) = PolynomialRing(QQ, ["x", "y", "z"])
       primary_decomposition(ideal(R, [z-x^2, z-y^3]))
2-element Vector{Oscar.MPolyIdeal}:
 ideal generated by: y^3 - z, x^2 - z
 ideal generated by: y^3 - z, x^2 - z
```

```
       Q, _ = quo(R, ideal(R, [z-x^2, z-y^3])); Q
Quotient of Multivariate Polynomial Ring in x, y, z over Rational
 Field by ideal generated by: -x^2 + z, -y^3 + z
```

very experimental interface for normalization:

```
julia> [(S, m, frac_ideal)] = normalize(Q); frac_ideal
(z, ideal generated by: x*y^2, z)
```

The integral closure of Q is generated by $\frac{xy^2}{z}$ and 1. (in Q, $\left(\frac{xy^2}{z}\right)^6 = z$)

Galois Groups

```
julia> Zx, x = ZZ["x"]; k, a = number_field(x^4 + 10*x^2 + 2);
       G, _ = galois_group(k); collect(G)
8-element Vector{Any}:
 ()
 (2,3)
 (1,4)
 (1,4)(2,3)
 (1,2)(3,4)
 (1,2,4,3)
 (1,3,4,2)
 (1,3)(2,4)
```

```
julia> using StatsBase; countmap([begin
           _, x = PolynomialRing(GF(p), "x"; cached = false)
           sort([degree(f) for (f, e) in factor(x^4+10x^2+2)])
       end
       for p in PrimesSet(20, 10^7)])
Dict{Vector{Int64}, Int64} with 5 entries:
  [2, 2]       => 249306    # 3/8 approx.
  [1, 1, 2]    => 166151    # 2/8
  [4]          => 166209    # 2/8
  [1, 1]       => 1
  [1, 1, 1, 1] => 82904     # 1/8
```

"generic" Factoring

```
julia> Za, a = ZZ["a"]; k, a = number_field(a^5 - 2)
       R, (x, y, z) = k["x", "y", "z"]
       AbstractAlgebra.obj_to_latex_string(factor(x^60-4*y^60*z^30))
```

$$1 \cdot \left(x^{24} - ax^{18}y^{6}z^{3} + a^{2}x^{12}y^{12}z^{6} - a^{3}x^{6}y^{18}z^{9} + a^{4}y^{24}z^{12}\right) \cdot$$
$$\left(x^{24} + ax^{18}y^{6}z^{3} + a^{2}x^{12}y^{12}z^{6} + a^{3}x^{6}y^{18}z^{9} + a^{4}y^{24}z^{12}\right) \cdot \left(x^{6} + ay^{6}z^{3}\right) \cdot \left(x^{6} - ay^{6}z^{3}\right)$$

very experimental absolute factorization:

```
julia> R, (x, y, z) = QQ["x", "y", "z"]; _, a = R["a"]
       factor_absolute(5*(x*y*z-1)^2*resultant(x+a*y+a^2*z,a^3+a+1))
5 * (x + (_a^2 - 2)*y + (_a^2 - _a - 2)*z)
 * (x^2 + (-_a^2 + 2)*x*y + (-_a^2 + _a)*x*z
     + (_a^2 - _a - 1)*y^2+ y*z + (-_a + 1)*z^2)
 * (x*y*z - 1)^2
```

```
julia> [base_ring(f) for (f, e) in ans]
3-element Vector{AbstractAlgebra.Field}:
 Number field over Rational Field with defining polynomial x^3 - 2*x^2 - x + 3
 Number field over Rational Field with defining polynomial x^3 - 2*x^2 - x + 3
 Rational Field
```

Code Samples

Many examples of using OSCAR and its components are here:
https://oscar.computeralgebra.de/example/

The code for factored elements is here:
https://gist.github.com/tthsqe12/73a17c5bb2182df8e9ff112e96d29606
In order to run this, it is necessary to have three packages installed:

```
julia> using Pkg; Pkg.add("Random"); Pkg.add("Test"); Pkg.add("Nemo")
```

Then it is a simple (or wherever the file is saved):

```
julia> include("FactoredElem.jl")
```

This will create the type and run some tests and examples.

Note that the field functionality here may be combined with the existing `FacElem` type in OSCAR in a near future release of AbstractAlgebra.jl.