Low-level libpari functions for finite fields

A guided tour

B. Allombert

Institut de Mathématiques de Bordeaux
Univ. Bordeaux, CNRS, INRIA

24/06/2025

Introduction to function families in libpari

While `GEN` can be quite complex, they are very few type numbers available. Thus the types only encodes the structure of the object but not any mathematical semantic.
To fix that, we created semantic labels that are used in libpari function prefix to denote the mathematical meaning of its arguments.
This makes the code more readable since we always know what is the meaning of the functions arguments, and it is easier to find them. However such functions never check their arguments, so it is always possible to substitute a `t_VEC` for a `t_COL` for example.
Sometime different functions can have identical implementation.

Function family: `Fp`

For p a `t_INT`, a `Fp` is a `t_INT` implicitely modulo p.
To reduce an integer x modulo p use `modii(x,p)`
For x, y, p `t_INT`, some members of this family:

- `GEN Fp_add(GEN x, GEN y, GEN p)` return $x + y \pmod{p}$
- `GEN Fp_sub(GEN x, GEN y, GEN p)` return $x - y \pmod{p}$
- `GEN Fp_mul(GEN x, GEN y, GEN p)` return $xy \pmod{p}$

etc.

Function family `Fl`

For p a `ulong`, a `Fl` is a `ulong` x implicitely modulo p. It is required that $x < p$.
To convert a `t_INT` x to a `Fl` use `umodiu(x,p)`.
For x, y, p `ulong` with $x < p$ and $y < p$, some members of this family:

- `ulong Fl_add(ulong x, ulong y, ulong p)` return $x + y \pmod{p}$
- `ulong Fl_sub(ulong x, ulong y, ulong p)` return $x - y \pmod{p}$
- `ulong Fl_mul(ulong x, ulong y, ulong p)` return $xy \pmod{p}$

Preconditionned reduction: `Fl..._pre`

To speed up reduction mod p (especially on intel CPU), we can compute a pseudo inverse `ulong pi=Fl_get_red(p);` and then use the `Fl..._pre` functions:

- `ulong Fl_mul_pre(ulong x, ulong y, ulong p, ulong pi)`
- `ulong Fl_sqr_pre(ulong x, ulong p, ulong pi)`

etc.

Function families ZV/ZC/ZM/FpV/FpC/FpM

ZV (resp. ZC) are t_VEC (resp. t_COL) with t_INT coefficients. ZM are matrices with t_INT coefficients.

Some members of these families:

- `GEN ZC_add(GEN x, GEN y)`
- `GEN ZM_mul(GEN x, GEN y)`
- `GEN ZM_ZC_mul(GEN x, GEN y)`

FpV, FpC, FpM idem but modulo p. Some members of these families:

- `GEN FpC_add(GEN x, GEN y, GEN p)`
- `GEN FpV_add(GEN x, GEN y, GEN p)`
- `GEN FpM_det(GEN x, GEN p)`

To reduce the coefficients modulo p use

- `GEN FpC_red(GEN x, GEN p)`
- `GEN FpV_red(GEN x, GEN p)`
- `GEN FpM_red(GEN x, GEN p)`

Function families `Flv/Flc/Flm`

`Flv` and `Flc` are identical, both are `t_VECSMALL`s with coefficients `uel(v,i)<p`.
`Flm` are `t_MAT` whose colums are actually `Flc`.

`Flm` are not well-supported by GP.

Some members of these families:

- `GEN Flv_add(GEN x, GEN y, ulong p)`
- `GEN Flm_mul(GEN x, GEN y, ulong p)`
- `GEN Flm_Flc_mul(GEN x, GEN y, ulong p)`
- `GEN Flm_mul_pre(GEN x, GEN y, ulong p, ulong pi)`
- `GEN Flm_Flc_mul_pre(GEN x, GEN y, ulong p, ulong pi)`

To convert `ZV/ZC/FpV/FpC` to `Flv/Flc` use

- `GEN ZV_to_Flv(GEN x, ulong p)`

To convert `ZM/FpM` to `Flm` use

- `GEN ZM_to_Flm(GEN x, ulong p)`

To lift `Flv/Flc/Flm` to `ZV/ZC/ZM` use

- `GEN Flv_to_ZV(GEN x)`
- `GEN Flc_to_ZC(GEN x)`
- `GEN Flm_to_ZM(GEN x)`

F2v/F2c

F2v and F2c are identical. They are t_VECSMALL storing a bit vector. v[1] is the dimension of the vector. The coefficients can be read or set using

- `ulong F2v_coeff(GEN x,long v)`
- `void F2v_set(GEN x,long v)`
- `void F2v_clear(GEN x,long v)`
- `void F2v_flip(GEN x,long v)`

To convert ZV/ZC/FpV/FpC to F2v/F2c use

- `GEN ZV_to_F2v(GEN x)`

To convert Flv/Flc F2v/F2c use

- `GEN Flv_to_F2v(GEN x)`

Function family F2m

F2m are t_MAT whose columns are actually F2c. The coefficients can be read or set using

- `ulong F2m_coeff(GEN x, long a, long b)`
- `void F2m_clear(GEN x, long a, long b)`
- `void F2m_set(GEN x, long a, long b)`
- `void F2m_flip(GEN x, long a, long b)`

To convert ZM/FpM (modulo 2) to F2m use

- `GEN ZM_to_F2m(GEN x)`

To convert Flm (modulo 2) to F2m use

- `GEN Flm_to_F2m(GEN x)`

Some members of these families:

- `GEN F2m_mul(GEN x, GEN y)`
- `GEN F2m_F2c_mul(GEN x, GEN y)`
- `GEN F2m_ker(GEN x)`

Function family ZX

ZX: t_POL whose elements are t_INT.
Some members of this family:

- GEN ZX_add(GEN x, GEN y, GEN p) return $x + y$
- GEN ZX_sub(GEN x, GEN y, GEN p) return $x - y$
- GEN ZX_mul(GEN x, GEN y, GEN p) return xy

Note that in that case we have signe(x)=0 iff lgpol(x)==0. x are assumed to be in the same variable which is not checked.

Function family FpX

FpX: t_POL whose elements are t_INT considered modulo p
To reduce a ZX modulo p use

- GEN FpX_red(GEN x, GEN p)

Some members of this family:

- GEN FpX_add(GEN x, GEN y, GEN p)
- GEN FpX_rem(GEN x, GEN y, GEN p)
- GEN FpX_Fp_mul(GEN x, GEN y, GEN p)

Function family `Flx`

`Flx` are `t_VECSMALL` encoding polynomials with `Fl` coeffients.

- `degpol(x)`: degree of x (-1 if $x = 0$), `degpol(x)=lg(x)-3`.
- `lgpol(x)`: `1+degpol(x)`, `lg(x)-2`.
- `signe(x)` is not defined. $x = 0$ iff $lgpol(x) = 0$.
- `x[1]`: shifted variable number
- `Flx_lead(x)`: leading coefficient.
- `Flx_constant`: constant coefficient.
- `pol0_Flx(sv)`, `pol1_Flx(sv)`, `polx_Flx(sv)` 0, 1, and x in shifted variable sv.

If P is of degree d, the coefficients of degree $0 \leq i \leq d$ can be accessed with `uel(P,i+2)`, so that `uel(P,i+2)<p` and `uel(P,d+2)` must not be 0.

Some members of this family:

- `GEN Flx_add(GEN x, GEN y, ulong p)`
- `GEN Flx_rem(GEN x, GEN y, ulong p)`
- `GEN Flx_rem_pre(GEN x, GEN y, ulong p, ulong pi)`
- `GEN Flx_Fl_mul(GEN x, ulong y, ulong p)`
- `GEN Flx_Fl_mul_pre(GEN x, ulong y, ulong p, ulong pi)`

To convert a `ZX/FpX` to a `Flx` use

- `GEN ZX_to_Flx(GEN x, ulong p)`

To lift a `Flx` to a `ZX` use

- `GEN Flx_to_ZX(GEN x)`

Function family F2x

F2x are t_VECSMALL encoding polynomials with $\{0, 1\}$ coeffients.

- F2x_degree(x): degree of x (-1 if $x = 0$).
- x[1] shifted variable number.
- signe(x) is not defined. $x = 0$ iff $lgpol(x) = 0$.
- pol0_F2x(sv), pol1_F2x(sv), polx_F2x(sv) 0, 1, and x in shifted variable sv.

The coefficients can be read or set using

- ulong F2x_coeff(GEN x,long v)
- void F2x_set(GEN x,long v)
- void F2x_clear(GEN x,long v)
- void F2x_flip(GEN x,long v)

Some members of this family:

- `GEN F2x_add(GEN x, GEN y)`
- `GEN F2x_rem(GEN x, GEN y)`

Some conversion functions:

- `GEN ZX_to_F2x(GEN x)`
- `GEN Flx_to_F2x(GEN x)`
- `GEN F2x_to_ZX(GEN x)`
- `GEN F2x_to_Flx(GEN x)`

Family function: FpXQ

FpXQ are FpX modulo some FpX T. They denote elements of the quotient ring $\mathbb{Z}/p\mathbb{Z}[X]/(T)$.
To reduce a ZX to a FpXQ use
`GEN FpXQ_red(GEN x, GEN T, GEN p)`
Some members of these families:

- `GEN FpXQ_mul(GEN x, GEN y, GEN T, GEN p)`
  return xy modulo T
- `GEN FpXQ_sqr(GEN x, GEN T, GEN p)`
  return x^2 modulo T

Precomputed inverse for FpXQ

To speed up modular reduction, it is possible to replace T by a precomputed inverse, by doing `T = FpX_get_red(T, p);` which accepted as a remplacement for T by `FpX_rem`. However, as a consequence, when implementing new `FpXQ` functions, T should not be accessed directly. The following accessors are provided

- `get_FpX_degree(T)`: return the degree of the original T.
- `get_FpX_var(T)`: return the variable number of the original T.
- `get_FpX_mod(T)`: (rarely needed) return the original T.

Family Fq

Fq are similar to FpXQ except that Fq can also be t_INT, in which case T can be NULL, to denote Fp.
Some member of the family:

- `GEN Fq_mul(GEN x, GEN y, GEN T, GEN p)`
- `GEN Fq_sqr(GEN x, GEN T, GEN p)`

Flxq/F2xq

Flxq are Flx modulo some Flx T. They are required to be of degree less than T.
F2xq are F2x modulo some F2x T. They are required to be of degree less than T.
Some examples:

- `GEN Flxq_mul(GEN x, GEN y, GEN T, ulong p)`
- `GEN Flxq_sqr_pre(GEN x, GEN T, ulong p, ulong pi)`
- `GEN Flx_Flxq_eval(GEN x, GEN y, GEN T, ulong p)`
- `GEN Flx_Flxq_eval_pre(GEN x, GEN y, GEN T, ulong p, ulong pi)`
- `GEN F2xq_mul(GEN x, GEN y, GEN T)`
- `GEN F2x_F2xq_eval(GEN x, GEN y, GEN T)`

Precomputed inverse for `Flxq`

To speed up modular reduction, it is possible to replace T by a precomputed inverse, by doing `T = Flx_get_red(T, p);` or `T = Flx_get_red_pre(T, p, pi);` which accepted as a remplacement for T by `Flx_rem` and `Flx_rem_pre`. However, as a consequence, when implementing new `Flxq` functions, T should not be accessed directly. The following accessors are provided

- `get_Flx_degree(T)`: return the degree of the original T.
- `get_Flx_var(T)`: return the shifted variable number of the original T.
- `get_Flx_mod(T)`: (rarely needed) return the original T.

Family functions: FqC/FqV/FqM

FqC/FqV/FqM are vectors/matrices with Fq coefficients.
To reduce a FqC/FqV/FqM use

- `GEN FqC_red(GEN x, GEN T, GEN p)`
- `GEN FqV_red(GEN x, GEN T, GEN p)`
- `GEN FqM_red(GEN x, GEN T, GEN p)`

Family functions: FlxqC/FlxqV/FlxqM/F2xqC/F2xqV/F2xqM

FlxqC/FlxqV/FlxqM are vectors/matrices with Flxq coefficients.
F2xqC/F2xqV/F2xqM are vectors/matrices with F2xq coefficients.
Some examples:

- `GEN FlxqM_FlxqC_mul(GEN x, GEN y, GEN T, ulong p)`
- `GEN FlxqM_ker(GEN x, GEN T, ulong p)`
- `GEN F2xqM_F2xqC_mul(GEN x, GEN y, GEN T)`
- `GEN F2xqM_ker(GEN x, GEN T)`

Conversions:

- `GEN FqC_to_FlxqC(GEN x, GEN T, ulong p)`
- `GEN FqM_to_FlxqM(GEN x, GEN T, ulong p)`
- `GEN FlxC_to_ZXC(GEN x)`
- `GEN FlxM_to_ZXM(GEN x)`
- `GEN F2xC_to_ZXC(GEN x)`
- `GEN F2xM_to_ZXM(GEN x)`

Family functions: FpXQX, FqX, FlxqX, F2xqX

FpXQX are t_POL whose coefficients are either t_INT or FpXQ of degree less than T. FqX are similar except that when all coefficients are t_INT, we allow T to be NULL.
FlxqX are t_POL whose coefficients are Flxq. F2xqX are t_POL whose coefficients are F2xq.
Some members of this family:

- `GEN FpXQX_rem(GEN x, GEN y, GEN T, GEN p)`
- `GEN FqX_rem(GEN x, GEN y, GEN T, GEN p)`
- `GEN FlxqX_rem(GEN x, GEN y, GEN T, ulong p)`
- `GEN F2xqX_rem(GEN x, GEN y, GEN T)`

Conversion functions

- To convert a `FpXQX` to a `FlxqX`, with `v=varn(T)`, use
  `GEN ZXX_to_FlxX(GEN B, ulong p, long v)`
- To convert a `FpXQX` to a `F2xqX`, with `v=varn(T)`, use
  `GEN ZXX_to_F2xX(GEN B, long v)`
- To convert a `FlxqX` to a `F2xqX`, use
  `GEN FlxX_to_F2xX(GEN B)`
- To convert a `FlxqX` to a `FpXQX`, use
  `GEN FlxX_to_ZXX(GEN B)`
- To convert a `F2xqX` to a `FpXQX`, use
  `GEN F2xX_to_ZXX(GEN B)`
- To convert a `F2xqX` to a `FlxqX`, use
  `GEN F2xX_to_FlxX(GEN B)`

Family functions: `FpXQXQ`, `FqXQ`, `FlxqXQ`, `F2xqXQ`

- `FpXQXQ` are `FpXQX` modulo some `FpXQX` S.
- `FqXQ` are `FqX` modulo some `FqX` S.
- `FlxqXQ` are `FlxqX` modulo some `FlxqX` S.
- `F2xqXQ` are `F2xqX` modulo some `F2xqX` S.

Some members of these families:

- `GEN FpXQXQ_mul(GEN x, GEN y, GEN S, GEN T, GEN p)`
- `GEN FqXQ_mul(GEN x, GEN y, GEN S, GEN T, GEN p)`
- `GEN FlxqXQ_mul(GEN x, GEN y, GEN S, GEN T, ulong p)`
- `GEN FlxqXQ_mul_pre(GEN x, GEN y, GEN S, GEN T, ulong p, ulong pi)`
- `GEN F2xqXQ_mul(GEN x, GEN y, GEN S, GEN T)`

Precomputed inverse for `FpXQXQ`

To speed up modular reduction, it is possible to replace S by a precomputed inverse, by doing `S = FpXQX_get_red(S, T, p);` which accepted as a remplacement for S by `FpXQX_rem`. However, as a consequence, when implementing new `FpXQXQ` functions, S should not be accessed directly. The following accessors are provided

- `get_FpXQX_degree(S)`: return the degree of the original S.
- `get_FpXQX_var(S)`: return the variable number of the original S.
- `get_FpXQX_mod(S)`: (rarely needed) return the original S.

Precomputed inverse for `FlxqXQ`

To speed up modular reduction, it is possible to replace S by a precomputed inverse, by doing `S = FlxqX_get_red(S, T, p);` or `S = FlxqX_get_red_pre(S, T, p, pi);` which accepted as a remplacement for S by `FlxqX_rem` and `FlxqX_rem_pre`. However, as a consequence, when implementing new `FlxqXQ` functions, S should not be accessed directly. The following accessors are provided

- `get_FlxqX_degree(S)`: return the degree of the original S.
- `get_FlxqX_var(S)`: return the variable number of the original S.
- `get_FlxqX_mod(S)`: (rarely needed) return the original S.

Precomputed inverse for `F2xqXQ`

To speed up modular reduction, it is possible to replace S by a precomputed inverse, by doing `S = F2xqX_get_red(S, T);` which accepted as a remplacement for S by `F2xqX_rem`. However, as a consequence, when implementing new `F2xqXQ` functions, S should not be accessed directly. The following accessors are provided

- `get_F2xqX_degree(S)`: return the degree of the original S.
- `get_F2xqX_var(S)`: return the variable number of the original S.
- `get_F2xqX_mod(S)`: (rarely needed) return the original S.

Families FpE/Fle/FpXQE/FlxqE/F2xqE

`FpE/FpXQE/FlxqE/F2xqE` are `t_VEC` with 2 components denoting affine points over an elliptic curve over $\mathbb{F}_p$ in short Weierstrass form $y^2 = x^3 + a_4x + a_6$ (in char. 2 and 3 different forms are used). `Fle` are `t_VECSMALL` with 2 component. In both cases the point at infinity is `ellinf` (use `ell_is_inf(P)` to test for it). Given a finite point on the curve, the curve is determined by a_4.

- `GEN FpE_add(GEN P, GEN Q, GEN a4, GEN p)`
- `GEN Fle_dbl(GEN P, ulong a4, ulong p)`
- `GEN FpXQE_order(GEN P, GEN Q, GEN o, GEN a4, GEN p)`
- `GEN FlxqE_weilpairing(GEN P, GEN Q, GEN m, GEN a4, ulong p)`
- `GEN F2xqE_tatepairing(GEN P, GEN Q, GEN m, GEN a2, GEN T)`

FF/FFX/FFM/FFE functions

The type `t_FFELT` is a wrapper around `FpXQ/Flxq/F2xq` objects in the file `src/basemath/FF.c`, the `FF` functions are build by applying the corresponding `FpXQ/Flxq/F2xq` function on the underlying objects, the `FFX` functions are build by applying the corresponding `FpXQX/FlxqX/F2xqX`, the `FFM` functions are build by applying the corresponding `FpXQM/FlxqM/F2xqM`, and the `FFE` functions are build by applying the corresponding `FpXQE/FlxqE/F2xqE`.

Example: FF_mul

```
GEN
FF_mul(GEN x, GEN y)
{
  ulong pp;
  GEN r, T, p, z=_initFF(x,&T,&p,&pp);
  pari_sp av=avma;
  _checkFF(x,y,"*");
  switch(x[1])
  {
  case t_FF_FpXQ:
    r=FpXQ_mul(gel(x,2),gel(y,2),T,p);
    break;
```

```
  case t_FF_F2xq:
    r=F2xq_mul(gel(x,2),gel(y,2),T);
    break;
  default:
    r=Flxq_mul(gel(x,2),gel(y,2),T,pp);
  }
  return _mkFF(x,z,gc_upto(av, r));
}
```

Function family Rg

Rg is the set of GP objects under GP operations +,-,*,/,% etc.
There is an expectation that Rg does not include components of type t_LIST, t_STR, t_VECSMALL, t_CLOSURE, t_ERROR and t_INFINITY.
It is not associative nor commutative even modulo == (gequal).
The name of the operations in C are gadd, gsub, gmul, gdiv, grem.

- int Rg_is_Fp(GEN x, GEN *pp) checks whether x can be converted to a Fp (modulo *pp if *pp is not NULL). If yes, set *pp to p if p can be determined.
- int Rg_is_FpXQ(GEN x, GEN *pT, GEN *pp) checks whether x can be converted to a FpXQ (modulo *pT,*pp if *pT,*pp are not NULL). If yes, set *pT to T and *pp to p if they can be determined.

Conversion from Rg

- `GEN Rg_to_Fp(GEN x, GEN p)` convert x to a (`t_INT`) `Fp` (modulo p).
- `ulong Rg_to_Fl(GEN x, ulong p)` convert x to a `Fl` (modulo p).
- `ulong Rg_to_F2(GEN x)` convert x to 0 or 1 (modulo 2).
- `ulong Rg_to_FpXQ(GEN x, GEN T, GEN p)` convert x to a `FpXQ` (modulo T, p).
- `ulong Rg_to_Fq(GEN x, GEN T, GEN p)` convert x to a `Fq` (modulo T, p if T is not `NULL`, modulo p otherwise).
- `ulong Rg_to_Flxq(GEN x, GEN T, ulong p)` convert x to a `Flxq` (modulo T, p).
- `GEN Rg_to_F2xq(GEN x, GEN T)` convert x to a `F2xq` (modulo `T`).

Function family RgC/RgV/RgM

RgC denotes t_COL, RgV denotes t_VEC and RgM denotes t_MAT.
Examples:

- `GEN RgV_dotproduct(GEN x)`
- `GEN RgM_RgC_mul(GEN x, GEN y)`
- `GEN RgM_mul(GEN x, GEN y)`
- `GEN RgM_inv(GEN x)`

Conversions

- `GEN RgC_to_FpC(GEN x, GEN p)`
- `GEN RgC_to_FqC(GEN x, GEN T, GEN p)`
- `GEN RgV_to_FpV(GEN x, GEN p)`
- `GEN RgV_to_FqV(GEN x, GEN T, GEN p)`
- `GEN RgV_to_Flv(GEN x, ulong p)`
- `GEN RgV_to_F2v(GEN x)`
- `GEN RgM_to_FpM(GEN x, GEN p)`
- `GEN RgM_to_FqM(GEN x, GEN T, GEN p)`
- `GEN RgM_to_Flm(GEN x, ulong p)`
- `GEN RgM_to_F2m(GEN x)`

Function family RgX

RgX denotes t_POL objects.
Identification functions:

- `int RgX_is_ZX(GEN x)` returns 1 if x is a ZX, 0 otherwise.
- `int RgX_is_FpX(GEN x, GEN *pp)` checks whether x can be converted to a FpX (modulo *pp if *pp is not NULL). If yes, set *pp to p if p can be determined.
- `int RgX_is_FpXQ(GEN x, GEN *pT, GEN *pp)` checks whether x can be converted to a FpXQX (modulo *pT,*pp if *pT,*pp are not NULL). If yes, set *pT to T and *pp to p if they can be determined.

Conversions

- `GEN RgX_to_FpX(GEN x, GEN p)`
- `GEN RgX_to_Flx(GEN x, ulong p)`
- `GEN RgX_to_F2x(GEN x)`
- `GEN RgX_to_FpXQX(GEN x, GEN T, GEN p)`
- `GEN RgX_to_FqX(GEN x, GEN T, GEN p)`
- `GEN RgX_to_FlxqX(GEN x, GEN T, ulong p)`