Iterators, black-box structures and callbacks

A day at the zoo

B. Allombert

IMB
CNRS/Université de Bordeaux

06/01/2025

How to implement `forprime` in C

For p `t_INT`:

```
GEN fun(GEN a, GEN b)
{
  forprime_t T;
  GEN p;
  forprime_init(&T,a,b);
  while ((p=forprime_next(p))
  {
    ...
  }
}
```

For *p* ulong:

```
GEN fun(ulong a, ulong b)
{
  forprime_t T;
  ulong p;
  u_forprime_init(&T,a,b);
  while ((p=u_forprime_next(p))
  {
    ...
  }
}
```

forprimestep

For p `t_INT`:

```
GEN fun(GEN a, GEN b, GEN q)
{
  forprime_t T;
  GEN p;
  forprimenext_init(&T,a,b,q);
  while ((p=forprime_next(p))
  {
    ...
  }
}
```

forprimestep

For *p* ulong, q=Mod(c,d)

```
GEN fun(ulong a, ulong b, ulong c, ulong d)
{
  forprime_t T;
  GEN p;
  u_forprime_arith_init(&T,a,b,c,d);
  while ((p=forprime_next(p))
  {
    ...
  }
}
```

Other less useful iterators

forvec Type: `forvec_t`

```
int forvec_init(forvec_t *T, GEN x, long
flag)
GEN forvec_next(forvec_t *T)
```

forpart Type: `forpart_t`

```
void forpart_init(forpart_t *T, long k,
GEN abound, GEN nbound)
GEN forpart_next(forpart_t *T)
```

forperm Type: `forperm_t`

```
void forperm_init(forperm_t *T, GEN n)
GEN forperm_next(forperm_t *T)
```

forsubset Type: `forsubset_t`

```
void forsubset_init(forsubset_t *T, GEN n)
GEN forsubset_next(forsubset_t *T)
```

generic binary powering/ double and add algorithm

PARI provide function for generic binary powering, for $n \geq 1$

For `t_INT` n
```
gen_pow(GEN x, GEN n, void *E, GEN
        (*sqr)(void*,GEN), GEN
        (*mul)(void*,GEN,GEN))
```

For `ulong` n
```
gen_powu(GEN x, ulong n, void *E, GEN
         (*sqr)(void*,GEN), GEN
         (*mul)(void*,GEN,GEN))
```

These functions compute x^n in the monoid where the multiplication is given by $(a, b) \mapsto mul(E, a, b)$ and squaring is given by $a \mapsto sqr(E, a)$ where E is a user-defined data structure (E can even be `NULL`).

Example Fp_powu

For example Fp_powu could be implemented as

```
static GEN
_Fp_mul(void *E, GEN x, GEN y)
{
  GEN p = (GEN) E;
  return Fp_mul(x,y,p);
}
static GEN
_Fp_sqr(void *E, GEN x)
{
  GEN p = (GEN) E;
  return Fp_sqr(x,p);
}
```

```
GEN
Fp_powu(GEN x, ulong n, GEN p)
{
  if (n==0) return gen_1;
  return gen_pow(x,n,(void*)p, _Fp_sqr, _Fp_mul);
}
```

Please do not reimplement `double` and `add`.

Folded multiplication variants.

Variant where `mul` is replaced $msqr(E, a) = xa^2$. Useful for example if $x = 2$.

- `GEN gen_pow_fold(GEN x, GEN n, void *E, GEN (*sqr)(void*,GEN), GEN (*msqr)(void*,GEN))`
- `GEN gen_powu_fold(GEN x, ulong n, void *E, GEN (*sqr)(void*,GEN), GEN (*msqr)(void*,GEN))`

gen_powers

- `GEN gen_powers(GEN x, long l, int use_sqr, void *E, GEN (*sqr)(void*,GEN), GEN (*mul)(void*,GEN,GEN), GEN (*one)(void*))`
  return the vector $[one(), x, x^2, \ldots, x^l]$. If `use_sqr=1`, compute x^{2n} as $(x^n)^2$ instead of $x.x^{2n-1}$.
- `GEN gen_product(GEN x, void *E, GEN (*mul)(void *,GEN,GEN))`
  return the product of the coefficients of the vector x by applying `mul`, using divide-and-conquer strategy.

generic sorting

Several functions implement generic sorting (merge sort).

- `GEN gen_sort(GEN x, void *E, int (*cmp)(void*,GEN,GEN))` Sort x according to `cmp`: `cmp(E,a,b) geq 0` iff $a \geq b$
- `GEN gen_indexsort(GEN x, void *E, int (*cmp)(void*,GEN,GEN))` indirect sort (return a permutation as a `t_VECSMALL` (`perm` family)).
- `long gen_search(GEN x, GEN y, void *data, int (*cmp)(void*,GEN,GEN))` Assuming x is sorted according to `cmp`, search y in x. Return the index where y is, or minus the index where it should be inserted.

When E is not needed, `cmp_nodata` can be used as follow: `gen_sort(x, (void*)cmp, cmp_nodata)` with `int cmp(GEN x, y)`

Sorting without duplicates

- `GEN gen_indexsort_uniq(GEN x, void *E, int (*cmp)(void*,GEN,GEN))`
- `GEN gen_sort_uniq(GEN x, void *E, int (*cmp)(void*,GEN,GEN))`

Predefined functions for `t_VECSMALL`

- `void vecsmall_sort(GEN V)` (in place)
- `GEN vecsmall_indexsort(GEN V)`

Predefined functions for vectors of `t_INT`

- `GEN ZV_sort(GEN L)`
- `GEN ZV_indexsort(GEN L)`
- `GEN ZV_sort_uniq(GEN L)`

Generic hash table

Hashtable (which are different from maps, that use AVL trees) are of type `hashtable`.

- `ulong hash_GEN(GEN x)` universal hash function valid for all `GEN`.
- `void hash_init(hashtable *h, ulong minsize, ulong (*hash)(void*), int (*eq)(void*,void*), int use_stack)` Initialize `h` for a startin size `minsize`, using the hash function `hash` and the equality function. `eq`. If `use_stack=1` it is allocated on the stack, otherwise it needs to be freed with `hash_destroy`.
- `void hash_destroy(hashtable *h)` free `h` assuming `use_stack=0`.

- `void hash_insert(hashtable *h, void *k, void *v)` Insert key k with value v.
- `hash_insert_long(hashtable *h, void *k, long v)` Idem for v of type `long`.
- int hash_haskey_long(hashtable *h, void *k, long *v) If k is in h return 1 and set v to the value, otherwise return 0.
- GEN hash_haskey_GEN(hashtable *h, void *k) If k is in h return the value otherwise return `NULL`.
- GEN hash_keys_GEN(hashtable *h) Return the list of keys as a `t_VEC`.

black box groups

A black box group is defined by a `struct bb_group`

```
struct bb_group
{
  GEN   (*mul)(void *E, GEN, GEN);
  GEN   (*pow)(void *E, GEN, GEN);
  GEN   (*rand)(void *E);
  ulong (*hash)(GEN);
  int   (*equal)(GEN,GEN);
  int   (*equal1)(GEN);
  GEN   (*easylog)(void *E, GEN, GEN, GEN);
};
```

(where `easylog` can be `NULL`).

Some operations on black box group

(in the following o can be an `t_INT` or a pair $[o, factor(o)]$)

- `GEN gen_gener(GEN o, void *E, const struct bb_group *grp)` return a generator of the group assuming it is cyclic of order o.
- `GEN gen_order(GEN x, GEN o, void *E, const struct bb_group *grp)` return the order of x assuming that it divides o.
- `GEN gen_select_order(GEN o, void *E, const struct bb_group *grp)` Assuming the group is cyclic and is an element of the `t_VEC` o, find it.

- `GEN gen_PH_log(GEN a, GEN g, GEN o, void *E, const struct bb_group *grp)` compute the discrete logarithm of a in base g, assuming g is of order o, using Pohlig-Hellman algorithm (and `easylog` if available).
- `GEN gen_Shanks_sqrtn(GEN a, GEN n, GEN o, GEN *zetan, void *E, const struct bb_group *grp)` Compute the n-roots of a in the cyclic group of order o, using Shanks algorithm.

Predefined black box groups

Some groups are predefined.

To define $\mathbb{F}_p[X]/T)^\times$, use

- `const struct bb_group * get_FpXQ_star(void **E, GEN T, GEN p)`
- `const struct bb_group * get_Flxq_star(void **E, GEN T, ulong p)`

to be used this way:

```
GEN
Flxq_order(GEN a, GEN ord, GEN T, ulong p)
{
  void *E;
  const struct bb_group *S = get_Flxq_star(&E,T,p);
  return gen_order(a,ord,E,S);
}
```

group of points on an elliptic curve

To define the group of points on an elliptic curve use

- `const struct bb_group * get_FpE_group(void **E, GEN a4, GEN a6, GEN p)`
- `const struct bb_group * get_FpXQE_group(void **E, GEN a4, GEN a6, GEN T, GEN p)`
- `const struct bb_group * get_FlxqE_group(void **E, GEN a4, GEN a6, GEN T, ulong p)`
- `const struct bb_group * get_F2xqE_group(void **E, GEN a2, GEN a6, GEN T)`

Black box fields

A black box field is defined by a `struct bb_field`

```
struct bb_field
{
  GEN (*red)(void *E ,GEN);
  GEN (*add)(void *E ,GEN, GEN);
  GEN (*mul)(void *E ,GEN, GEN);
  GEN (*neg)(void *E ,GEN);
  GEN (*inv)(void *E ,GEN);
  int (*equal0)(GEN);
  GEN (*s)(void *E, long);
};
Non canonical forms are allowed, and are normalized with \
```

Some functions for black box field

- `GEN gen_matmul(GEN a, GEN b, void *E, const struct bb_field *ff)`
- `GEN gen_det(GEN a, void *E, const struct bb_field *ff)`
- `GEN gen_ker(GEN x, long deplin, void *E, const struct bb_field *ff)`
- `GEN gen_matinvimage(GEN a, GEN b, void *E, const struct bb_field *ff)`
- `GEN gen_Gauss(GEN a, GEN b, void *E, const struct bb_field *ff)`

predefined blackbox fields

- `const struct bb_field *get_Fp_field(void **E, GEN p)`
- `const struct bb_field *get_Fq_field(void **E, GEN T, GEN p)`
- `const struct bb_field *get_Flxq_field(void **E, GEN T, ulong p)`
- `const struct bb_field *get_F2xq_field(void **E, GEN T)`
- `const struct bb_field *get_nf_field(void **E, GEN nf)`

Example

```
GEN
FqM_det(GEN x, GEN T, GEN p)
{
  void *E;
  const struct bb_field *S =
                get_Fq_field(&E,T,p);
  return gen_det(x, E, S, _FqM_mul);
}
```

blackbox algebra

A black box algebra is defined by a `struct bb_algebra`

```
struct bb_algebra
{
  GEN (*red)(void *E, GEN x);
  GEN (*add)(void *E, GEN x, GEN y);
  GEN (*sub)(void *E, GEN x, GEN y);
  GEN (*mul)(void *E, GEN x, GEN y);
  GEN (*sqr)(void *E, GEN x);
  GEN (*one)(void *E);
  GEN (*zero)(void *E);
};
```

Non canonical forms are allowed, and are normalized with `red`.

Example functions

- `GEN gen_bkeval(GEN Q, long d, GEN x, int use_sqr, void *E, const struct bb_algebra *ff, GEN cmul(void *E, GEN P, long a, GEN x))`
- `GEN gen_bkeval_powers(GEN P, long d, GEN V, void *E, const struct bb_algebra *ff, GEN cmul(void *E, GEN P, long a, GEN x))` (here `P` is a blackbox polynomial).

Predefined bb_algebra

- `const struct bb_algebra * get_FpXQ_algebra(void **E, GEN T, GEN p)`
- `const struct bb_algebra * get_FpXQXQ_algebra(void **E, GEN S, GEN T, GEN p)`
- `const struct bb_algebra * get_FlxqXQ_algebra(void **E, GEN S, GEN T, ulong p)`
- `const struct bb_algebra * get_Rg_algebra(void)`

More predefined bb_algebra

- `const struct bb_algebra * get_FpX_algebra(void **E, GEN p, long v)`
- `const struct bb_algebra * get_FpXQX_algebra(void **E, GEN T, GEN p, long v)`
- `const struct bb_algebra * get_FlxqX_algebra(void **E, GEN T, ulong p, long v)`

Example

This example use black-box polynomial.

```
static GEN
_FpXQXQ_cmul(void *data, GEN Pp, long a, GEN x)
{
  GEN P = gel(Pp,1), p = gel(Pp,2);
  GEN y = gel(P,a+2);
  return typ(y)==t_INT ? FpXX_Fp_mul(x,y, d->p):
                         FpXX_FpX_mul(x,y,d->p);
}
```

```
GEN
FpXQX_FpXQXQ_eval(GEN Q, GEN x,
                  GEN S, GEN T, GEN p)
{
  void *E;
  const struct bb_algebra *A =
        get_FpXQXQ_algebra(&E, S, T, p);
  int use_sqr =
      2*degpol(x) >= get_FpXQX_degree(S);
  return gen_bkeval(mkvec2(Q,p), degpol(Q), x,
         use_sqr, (void*)E, A, _FpXQXQ_cmul);
}
```

black box ring

A black box ring is defined by a `struct bb_ring`

```
struct bb_ring
{
  GEN (*add)(void *E, GEN x, GEN y);
  GEN (*mul)(void *E, GEN x, GEN y);
  GEN (*sqr)(void *E, GEN x);
};
```

- `GEN gen_fromdigits(GEN x, GEN B, void *E, struct bb_ring *r)`
- `GEN gen_digits(GEN x, GEN B, long n, void *E, struct bb_ring *r, GEN (*div)(void *E, GEN x, GEN y, GEN *r))`