Closures and parallelism

A cruise on the moat

B. Allombert

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

24/06/2025

`t_CLOSURE`

`t_CLOSURE` holds GP functions.
The length (`lg(C)`) can be $6, 7$ or 8.

- inline closure: 6
- function: 7
- true closure: 8

`closure_arity(C)`: arity of the closure.

True closures are GP functions that have a non empty context of execution:

```
? my(z=3);trueclosure(x)=x+z
%1 = (x)->my(z=3);x+z
```

Inline closure is code that appear inside loop:

```
? for(i=1,100,print(i^2+1))
```

`print(i^2+1)` is an inline closure (that depend on *i*).

Associating entree* to C functions

For GP to be able to call a C function, the C function need to have an `entree*`. There are three way to create it:

- use `install` in GP.
- use `pari_add_function` between `pari_init()` and `pari_mt_init()`; see `examples/pari-mt.c`.
- add a function description in `src/functions/`

In each case, three data are needed

- the name of the `entree*`.
- the name of the C function associated to it.
- the prototype code definin the GP interface to the C function.

If the name of the `entree*` is a valid GP variable name, the C function will be available under GP under that name. It is customary to prefix private functions name with `_`.

Prototype codes

The prototype code is as follow: if the first letter is one of `vluim` the return type is

- `v`: `void`
- `l`: `long`
- `u`: `ulong`
- `i`: `int`
- `m`: incomplete `GEN`

otherwise the return type is `GEN`.

Codes for argument

Then a code is added for each argument of the C function in order:

- G: `GEN`
- DG: `GEN` or `NULL`
- L: `long`
- U: `ulong`
- s: `const char *`
- n: `long` variable number
- p: the precision (`prec`)
- V: inline variable for inline closure
- I: inline closure returning `void`
- E: inline closure returning `GEN`
- J: closure of arity 1 for parallel code

See ??prototype for more detail.

Example

Under GP, do
`install("gadd",GG,"add")`
to define a GP function `add` that call `gadd`.
Or add a file `src/functions/programming/add` with

```
Function: add
C-Name: gadd
Prototype: GG
Section: programming/internals
Help: addition worker
```

and rebuild PARI.

Creating closure in C

- To convert GP text to a `t_CLOSURE` do `gp_read_str("(x)->my(z=3);x+z")`.
- To create `t_CLOSURE` from a `entree*`, use `strtofunction` or `strtoclosure` for true closure.

```
? install(strtofunction,s)
? install(strtoclosure,sLDGDG)
? s=strtofunction("_+_")
%3 = _+_
? s=strtoclosure("_+_",1,5)
%4 = (v1)->_+_(v1,5)
? s=strtoclosure("_+_",2,3,4)
%5 = ()->_+_(3,4)
? s()
%6 = 7
```

Calling closure in C

For a closure returning a `GEN`, of arity $0, 1, 2, \ldots$:

- `GEN closure_callgen0(GEN C)`
- `GEN closure_callgen1(GEN C, GEN x)`
- `GEN closure_callgen2(GEN C, GEN x, GEN y)`
- `GEN closure_callgenvec(GEN C, GEN args)`

For a closure without return value, of arity 1.

- `void closure_callvoid1(GEN C, GEN x)`

For a closure under `localbitprec(prec)`:

- `GEN closure_callgen0prec(GEN C, long prec)`
- `GEN closure_callgen1prec(GEN C, GEN x, long prec)`
- `GEN closure_callgenvecprec(GEN C, GEN args, long prec)`

Example: apply

```
GEN my_apply(GEN C, GEN V)
{
  long i, l = lg(V);
  GEN W = cgetg(l, t_VEC);
  for (i = 1; i < l; i++)
    gel(W, i) = closure_callgen1(C, gel(V,i));
  return W;
}
```

Inline closure in C

In the example: `matrix(4,5,i,j,i+j)`, the prototype code of `matrix` is `GDGDVDVDE` where the first `DV` is for the inline variable `i`, the second for `j` and `DE` is for the inline closure `i+j`.

- `void push_lex(GEN a, GEN C)`: push a new inline variable with value *a* (and number -1), where *C* is the inline closure, decreasing the number of the previously defined variables.
- `void set_lex(vn, a)`: set the preexisting inline variable with number *vn* to *a*.
- `void pop_lex(long n)`: pop the last *n* inline variables.

Inline closure in C

- `closure_evalvoid(C)`: call C, ignoring the return value.
- `closure_evalnobrk(C)`: call C, get the return value, disallow `break,next,return`.
- `closure_evalgen(C)`: call C, get the return value, allow `break,next,return`.
- `loop_break()`: check whether `break,next,return` happened.

Example

```
void forprime(GEN a, GEN b, GEN code)
{
  forprime_t T;
  GEN p;
  forprime_init(&T, a,b);
  push_lex(gen_0, code);
  while((p=forprime_next(&T)))
  {
    set_lex(-1,p);
    closure_evalvoid(code);
    if (loop_break()) break;
  }
  pop_lex(1);
}
```

closuretoinl

`closuretoinl(C): convert a closure to an pseudo-inline closure suitable for codes E and I`

Example:

```
? install("forprime","vV=GGI","myforprime1")
? myforprime1(p=2,10,print1(p," "))
2 3 5 7
? install("closuretoinl","G")
? install("forprime","vGGG","myforprime2")
? myforprime2(2,10,closuretoinl(p->print1(p," ")))
2 3 5 7
? my(z=3);myforprime2(2,10,closuretoinl(p->z+=p));z
%6 = 3
```

The catch is that the closure is executed in a new lexical scope.

Parallelism in libpari: parapply

To run code in a parallel section, it is necessary to embed it in a `t_CLOSURE` so that it can be send across the network with MPI. In libpari it is customary to call such private C function with the prefix `worker`, to prefix the GP name with `_` and to add the C prototype to `src/headers/paripriv.h`, and use the GP section `programming/internals`.

For example to use parapply with `GEN myfun_worker(GEN x, GEN c)` with *c* a user-specified parameter: add a file in `src/functions/` with

```
Function: _myfun_worker
C-Name: myfun_worker
Prototype: GG
Section: programming/internals
Help: worker for myfun
```

Calling parapply

Then we can convert it to a `t_CLOSURE` and call `parapply` on it:

```
GEN parmyfun(GEN D, GEN c)
{
  GEN worker = strtoclosure(
                 "_myfun_worker", 1, c);
  return parapply(worker, D);
}
```

Low level parallel interface

A more flexible, lower-level interface is available that finer control:

```
GEN parapply(GEN worker, GEN V)
{
  long i, l = lg(V), pending = 0;
  struct pari_mt pt;
  GEN W = cgetg(l, typ(V));
  mt_queue_start_lim(&pt, worker, l-1);
  for (i = 1; i < l || pending; i++)
  {
    long workid;
    GEN done, work = i<l? mkvec(gel(V,i)): NULL;
    mt_queue_submit(&pt, work);
    done = mt_queue_get(&pt, &workid, &pending);
    if (done)
      gel(W,workid) = done;
  }
  mt_queue_end(&pt); return V;
}
```

When using MPI, the worker is send only once to each nodes, while `mt_queue_submit` send `work` to a single node. One should take care to minimize data transfer.