gp2c types and the description system

By Bill Allombert

April 15, 2020

Contents

1 gp2c types

1.1 Introduction

The main feature GP2C adds above GP is the use of types. Types give a semantic to PARI objects, so that GP2C can generate code that use specialized (hence faster) PARI functions instead of generic ones. Please read the section 'Advanced use of GP2C' in the GP2C manual for how to use the GP2C types and examples.

Such types are used in conjunctions with so-called *descriptions* which are stored in the field 'Description:' of the file `pari.desc` and provide the actual

C code to use depending of the types of the arguments. They are described in Section 2.

Abstractly, a GP2C type is a set of pairs (A, B) where A is a mathematical object and B its computer representation. Two different types can contain the same mathematical object, with a different computer representation. For example the type *bool* is the set {(true,1L), (false,0L)}, *i.e.* true and false with true coded by the C-long integer 1 and false coded by the C-long integer 0; the type *negbool* is the set {(true,0L), (false,1L)} which is the same set of mathematical objects, but now true is coded by the C-long integer 0 and false by the C-long integer 1.

For each GP2C type, there exists a C type Ct such that for all pairs (A, B) belonging to the type, the C type of B is Ct. This C type is specified by the description _typedef.

The GP2C types are preordered. Abstractly, we say that $t_1 \prec t_2$ if and only if there is a map f such that $(A, B) \mapsto (A, f(B))$ defines a one-to-one mapping from t_1 to t_2. Practically we restrict the relation $\prec$ to make it a partial order such that any two types have an upper bound. This partial order is defined by the chains in the description _type_preorder. It can be printed by running `gp2c -t`.

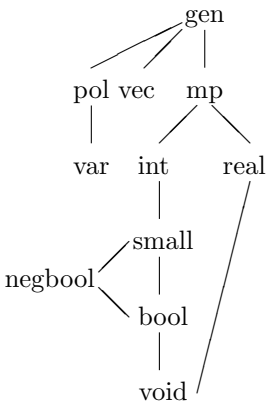

Figure 1: Example of type preorder

The process of converting a mathematical object from one type to another is called casting. The casting methods known to GP2C are given by the _cast description.

1.2 List of types

In this section, we list the types known to PARI/GP. The current list is available in the description _typedef.

1.2.1 Basic types

small Small integers represented by C long integers.

int Multi-precision integers represented by `t_INT` GENs.

real Multi-precision floating point real numbers represented by `t_REAL` GENs.

mp Multi-precision numbers. Union of the types *int* and *real*.

vecsmall Vectors of small integers represented by `t_VECSMALL` GENs.

vec Vectors and matrices of PARI objects, represented by `t_VEC`, `t_COL` or `t_MAT` GENs.

var Polynomial variables represented by their variable number which is a C long integer. This is associated to the prototype code 'n'.

pol Polynomials represented by `t_POL` GENs.

genstr Strings represented by `t_STR` GENs.

list GP lists represented by `t_LIST` GENs.

gen Generic PARI objects represented by GENs.

1.2.2 Special types

void This type is a set with only one element called *void*. This is the return type of functions not returning anything. GP allows to cast it to 0.

bool Boolean values represented as C long integers, where 1 is true and 0 is false.

negbool Negated boolean values represented as C long integers, where 0 is true and 1 is false.

lg Vector lengths represented by the lg macro output, i.e. a C long integer equal to the actual length plus one.

str C strings represented by C const char * strings.

typ GEN types represented by C long integers, as returned by the typ() macro.

1.2.3 Internal types

The following types are mainly for internal use inside GP2C.

empty This type is the empty set. No individual object can be of this type but a set of objects can. In fact this is a default type for an unspecified set of objects.

small_int Small integers represented by C int integers. This is only available for compatibility with PARI function returning int (instead of long).

bptr Byte pointer. This is used for the primepointer global variable.

func_GG function with protype GEN f(GEN,GEN). Used by *forvec*.

pari_sp This is the stack pointer type pari_sp.

1.2.4 Types for data structure.

These types are mainly defined to allow the use of inline member functions.

nf Number field as returned by *nfinit*.

bnf Big number field as returned by *bnfinit*.

bnr Ray class field as returned by *bnrinit*.

ell Elliptic curve as returned by *ellinit*.

bell Big elliptic curve as returned by *ellinit*.

clgp Class group as returned by the 'clgp' member function.

prid Prime ideal as returned by *idealprimedec*.

gal Galois group as returned by *galoisinit*.

1.3 C-Type

A *C-Type* is just a level of indirection toward real C types. C-types are defined by the descriptions _decl_base and _decl_ext. Each type belongs to a C-type as specified by the description _typedef.

2 The description system

2.1 Introduction

The description system is a way to describe the PARI application programming interface in a way understandable by both the GP2C compiler and human beings. The present document is mostly addressed to this second category. We start by a simple example:

The description of the GP function *sqr* is

```
(int):int       sqri($1)
(mp):mp         gsqr($1)
(gen):gen       gsqr($1)
```

Each line is called a *rule*, which in this case consists of three parts. Let us consider the first one: the parts `(int)`, `:int` and `sqri($1)` are respectively called the *pattern*, *type*, and *action* part.

When GP2C compiles *sqr(1)*, it computes the types of the arguments (here *1* is of type *small*) and matches them against the patterns from top to bottom. The "best" rule is used; in case of a tie, the topmost rule wins. Here, all three rules apply, and the first rule wins. Since the type of this rule is *int*, GP2C sets the type of the expression *sqr(1)* to *int*. The action part is `sqri($1)`, so GP2C generates the C code sqri($1) where $1 is replaced by the code of the argument *1* cast to the pattern type (*int*). The result is the C code sqri(gen_1).

Now a more complex example: the description of the GP function *exp* is

```
(real):real     mpexp($1)
(mp):mp:prec    gexp($1, prec)
(gen):gen:prec  gexp($1, prec)
```

When GP2C compiles *exp(1)*, it looks for the "best" rules. The first rule cannot be used, because there is no way to cast a *small* to a *real*, so it uses the second rule. This time the result will be of type *mp*. The extra part `:prec` is called a mode. The mode 'prec' states that the action part will use the special 'prec' variable that holds the current real precision. This is obvious from the action part code, but GP2C do not parse the action part so it needs this mode. Note that the last rule is also valid and has the same action part so would generate the exact same code. However, the type of the expression would be less precise.

The description of the GP function *matdet* is

```
(gen, ?0):gen      det($1)
(gen, 1):gen       det2($1)
(gen, #small):gen  $"incorrect flag in matdet"
(gen, small):gen   det0($1, $2)
```

We see several new pattern atoms:

- *1* matches a literal 1, e.g. *matdet(M,1)* would match the second rule.
- *?0* matches an optional literal 0: *matdet(M)*, *matdet(M,0)* and *matdet(M,)* all match the first rule.
- *#small* matches an unspecified literal *small*.

Finally, we also see a new action *$"..."*, which causes GP2C to display the error message and abort.

2.2 Definitions

We now give a formal definition of descriptions.

2.2.1 Description

A description is a line-separated list of *rules*.

2.2.2 Rule

A rule is a line of the form

(*pattern*)*:type:modelist action*

Only the pattern part is mandatory, though most rules also include an action and a type.

2.2.3 Pattern

A pattern is a comma-separated list of *pattern atoms*.

2.2.4 Type

The type of a rule is a standard GP2C type.

2.2.5 Modelist

A modelist is a colon-separated list of *modes*.

2.2.6 Action

An action is a string (normally a piece of C code) that can include *replacement strings*. Replacement strings start by a $ and are substituted according to the replacement rules.

2.3 Pattern atom

A pattern atom is one of the following, where *type* is any GP2C type, *n* any small integer, *"str"* any character string and *ctype* any C-type. A pattern atom can match an object.

- *type*. This matches any object of type comparable to *type*.
- *n*. This matches a constant small integer value equal to *n*.
- *?n*. This matches an optional *small* value which defaults to *n*.
- *?type*. This matches an optional *type* value with standard default value.
- *"str"*. This matches a constant character string equal to *str*.
- *&type*. This matches a reference (the GP *&x* construction) to an object of type equal or less than *type* referencing the same data type.
- *nothing*. This matches a missing argument.
- *#type*. This matches a constant value of type *type*.
- *...* This matches any number of arguments matching the previous atom. This must be the last atom of the pattern. This allows to implement functions taking an unlimited number of arguments.
- *C!ctype*. This matches any object of C-type *ctype*.
- *@type*. This matches a variable of type *type*. This is mainly used for expressions that evaluate their arguments several times.
- **type*. This matches an lvalue of type *type*. This is used in constructions that modify their arguments.

2.4 Matching

The best rule is determined as follows:

1. The result of matching a pattern atom against some GP code is either 'reject' or 'match'.
2. There are three matching levels: 'partial', 'normal' and 'perfect'.
3. A pattern matches if all the atoms match.
4. A rule matches if its pattern matches.
5. The best rule is the matching rule with the higher number of normal and perfect matches. In case of a tie, the highest number of perfect matches wins. If there is still a tie, the topmost rule wins.

When matching the pattern atoms *type* and *?type*, the matching level is determined as follows:

- a perfect match occurs when the type of the object is exactly *type*,
- a normal match when the type is less than *type*,
- a partial match when the type is bigger than *type*.
- Rejection happens when the types are uncomparable.

Other pattern atoms always result in a reject or a perfect match.

2.5 Mode

Modes are used in descriptions to give more information to GP2C about the action part. They are usually useless to human beings that are smart enough to understand the action part. The current list of modes is:

prec The action uses the prec variable.

parens The action does not have top precedence. GP2C will put it between parentheses when needed (see $())

copy The action returns data that access memory belonging to other objects. GP2C will generate calls to gcopy() when needed.

2.6 Lists of replacement strings

The following special sequences can occur in the action part:

- $n. This is replaced by the n-th argument of the function.
- $(n). This is replaced by the n-th argument of the function between parenthesis if it has the parens mode.
- $type:n. This is replaced by the n-th argument of the function cast to type *type*.
- $(type:n). Combination of $(n) and $type:n.
- $%n. This is replaced by the n-th argument of the function, which must be a constant string, with all % characters doubled and no quotes. This is for use inside format specification.
- $prec: short cut for $prec.
- $bitprec: short cut for $bitprec.
- $"message". Signals an invalid condition. GP2C will abort by printing the error message message.
- ${RPN sequence} The RPN sequence is a space separated list of RPN commands that will be evaluated by the GP2C internal RPN evaluator. If the stack is empty at the end of the evaluation, this is replaced by the empty string, else this is replaced by the integer at the top of the stack. Some RPN commands generate text, in that case it is pasted just before the $ sign.

2.7 Lists of RPN commands

The commands are evaluated with respect to a stack of integer values, initially empty. The exact list of command supported by a particular GP2C version is the `%accepted_command` hash in the script `scripts/822_desc.pl.in`.

literal integer push the integer at the top of the stack.

:type push the type *type* at the top of the stack.

add, sub, mul, div, mod 2-ary arithmetic operators

neg 1-ary arithmetic operator

and, or, xor 2-ary logical operators

not 1-ary logical operator

type pop an integer n and push the type of the n-th argument.

value pop an integer n and push the value of the n-th argument, provided it is a constant integer.

code pop an integer n and generate the C code for the n-th argument.

cast pop an integer n and a type t and generate the C code for the n-th argument cast to type t.

parens this is a flag requesting **code** and **cast** to enclose the C code between parenthesis if the argument has the *parens* mode.

str_format, str_raw pop an integer n such that the n-th argument is a constant string and display the string without leading and ending ". Furthermore **str_format** will display the string in a way suitable for inclusion in a format string by quoting meta-characters.

prec display the local precision (in prec format).

bitprec display the local precision in bit.

The following RPN commands are useful with the *...* pattern atom to implement functions that take an unlimited number of arguments.

nbarg push the actual number of arguments of the function.

format_string, format_args pop an integer n such that the n-th argument corresponds to a *...* pattern atom and generate a format string and a list of arguments, see the description of the GP function *print*.

code, cast If the integer n corresponds to a *...* pattern atom, generate a comma-separated list of C code for the arguments $n-1$, n, $n+1$, ..., **nbarg**, by matching each argument against the $n-1$ pattern atom.

stdref this is a flag requesting **code** and **type** to prepend a '&' before each arguments.

The following RPN commands are useful to implement functions that take closures as arguments.

wrapper, cookie pop an integer n and generate a call to the wrapper (resp. the cookie associated to the wrapper) for the n-th argument. The wrapper generated depends on the wrapper prototype in the Wrapper field. The cookie is the list of local variables seen by the closure.