Introduction the the Singular sources

Hans Schönemann

October 22, 2014

Preface

This paper should introduce some concept and conventions of SINGULAR which are not connected to a specific routine/type.o

For details, consult the sources or the reference manual about that specific routine.

Ich wünschte sehr der Menge zu behagen,
Besonders weil sie lebt und leben läßt.
Die Pfosten sind, die Bretter aufgeschlagen,
Und jedermann erwartet sich ein Fest.
Sie sitzen schon mit hohen Augenbraunen
Gelassen da und möchten gern erstaunen.
Ich weiß, wie man den Geist des Volks versöhnt;
Doch so verlegen bin ich nie gewesen:
Zwar sind sie an das Beste nicht gewöhnt,
Allein sie haben schrecklich viel gelesen.
Wie machen wir's, daß alles frisch und neu
Und mit Bedeutung auch gefällig sei?
Denn freilich mag ich gern die Menge sehen,
Wenn sich der Strom nach unsrer Bude drängt,
Und mit gewaltig wiederholten Wehen
Sich durch die enge Gnadenpforte zwängt;
Bei hellem Tage, schon vor vieren,
Mit Stößen sich bis an die Kasse ficht
Und, wie in Hungersnot um Brot an Bäckertüren,
Um ein Billet sich fast die Hälse bricht.
(Goethe, Faust, Vorspiel auf dem Theater)

Chapter 1

Naming Conventions

1.1 Functions

The general rule for function names (and other global visible names) is the structure `<prefix><Name>`. The main part (and each new word within) starts with a capital letter. A list of some common prefixes:

algebra related types		
SINGULAR	C/C++	function prefix
`int`	`int`	-
`string`	`char *`	-
`intvec`	`intvec *`	iv
`intmat`	`intvec *`	iv
`number`	`number`	n, np, nl, na
`poly`	`poly`	p
`vector`	`poly`	p
`ideal`	`ideal`	id
`module`	`ideal`	id
`matrix`	`matrix`, sometimes also `ideal`	mp
`intvec`	`intvec`	iv
`intmat`	`intvec`	iv
`resolution`	`resolution`	-
`list`	`lists`	l
`ring`	`ring`	r
interpreter related types		
`package`	`package`	
expression, expression list	`leftv`	
interpreter related prefix		
		`ii ji, jj`

1.2 Macros

Macros which are used to avoid multiple inclusion by `cpp` should be derived from the file name and completely in capital letters. The same applies to macros for constant values.

Macros which are used as functions should follow the convention for function

names above.

1.3 File names

Names for files are restricted by the usual file systems restrictions: they must be unique even if all capital letters are converted to small letters.

1.4 Local Variables

Usually `i,j,k,l,m,n` are `int` counter, `f,g,p` are often used for polynomials. Within the interpreter, `u,v,w` are the arguments, `res` the result, `h` is usually a pointer for an interpreter variable.

Chapter 2

Types and constants

2.1 Standard Types

- `ADDRESS` `void*`
- `BOOLEAN` a small integer type which at most place contains `FALSE, TRUE` and sometimes some more. For speed/space reasons it is `int` on 64bit machines and `short` otherwise.
  (`sizeof(char)<=sizeof(BOOLEAN)<=sizeof(int)`)

2.2 Standard Constants

C++ does not distinguish these values, but Singular does (depending on the intended type):

- `0` for `short/int`
- `0L` for `long`
- `'\0'` for `char`
- `NULL` for pointer

Chapter 3

I/O

3.1 Input

`stdin` is exclusively used by the scanner. Other input must use the methods of `si_link`. (see `Inout::pause`).

3.2 Output

Beside `si_link` output to `stdout` is provided by `Print` (analogue to `printf`)), `PrintS(char *)`, warning should be printed by `Warn/WarnS`, errors by `Werror/WerrorS`.

Other data type have their own output routines which are bases (exclusively) on `Print/PrintS`.

3.3 String based output

Many output routines are also used for conversion to `string`:

- start a new buffer with `StringSetS(char *)`
- (optional) append stuff with `StringAppendS`
- destroy the buffer and return its contents (an omalloc allocated block) with `StringEndS`.

The nesting level of `StringSetS`/ `StringEndS` is limited to 8.

3.4 Output routines

All output routines for certain types have `Write` in them: `n_Write` (for number, to the buffer), `p_write` (for poly), `iiWriteMatrix` (for ideal/matrix/map/module). `rWrite` (for ring), etc.

Chapter 4

Memory management

Singular uses `omalloc` as memory manager, which is optimized for many small blocks of similar sizes (very low overhead). Another difference to malloc is the alignment: address are aligned to the size of a pointer, not the largest simple type. For such requirements use `omAllocAligned`. The content of the newly allocated block is undefined unless `omAlloc0` (or similar) is used: the content of these blocks is set to 0.

- `omalloc`: analogue to malloc
- `omAlloc(s)`: Singular's optimized version: requires $s! = 0$.
- `omfree`: analogue to free
- `omFree(p)`: Singular's optimized version: requires $p! = NULL$.
- `omFreeSize`: checks the size in the debug version, otherwise equivalent to `omFree`
- `omAllocBin(b)`: uses a special free list created by `omGetSpecBin`
- `omFreeBin(p,b)`: uses a special free list created by `omGetSpecBin`, but blocks allocated by omAllocBin may also be freed by `omFree/omFreeSize`
- `new/delete`: standard C++ memory operators are overloaded, must be used in pairs, cannot be mixed neither with `omalloc` nor `new[]/delete[]`
- `new[]/delete[]`: standard C++ memory operators are overloaded, must be used in pairs, cannot be mixed neither with `omalloc` nor `new/delete`

Chapter 5

Numbers

The structure pointed to by "coeffs" describes a commutative ring with 1 which could be used as coefficients for polynomials.

To create a new instance of `coeffs`, implement all of the mandatory properties/functions (and, depending of the properties, the needed optional ones) and register it via `nRegister(n_unknown,<InitCharFunc>)`. `coeff` stores now the description of this ring and all the operations, its members are of type `number` (they should be considered as hidden pointers).

The general convention for these functions for numbers is that they create new (number-) objects (keeping the arguments) - with the exception of the cfInp... routines which modify their first argument.

All public routines are prefixed by `n_` and require the ring description (of type `coeff`) as last argument.

Chapter 6

Polynomials

6.1 poly and ring

Polynomials form a polynomial ring, commutative or non-commutative, together with a monomial ordering (global, local or mixed). The coefficients (`number`) commute with all polynomial variables.

The description of the ring is stored by objects of type `ring`, their objects are of type `poly`.

`poly` represents polynomials as simple linked list of monomials, ordered by the corresponding monomial ordering. `NULL` represents the polynomial 0. Each monomial consists of a coefficient (of type `number`, a `next` pointer and the representation of the exponent vector. Its encoding depends on the monomial ordering: only `p_GetExp/p_GetExpV` can retrieve the exponents, only `p_SetExp/p_SetExpV` can set them. After a call resp. several call to `p_SetExp` one needs to call `p_Setm`.

The general convention for polynomial functions is that they create new (poly-) objects by absorbing the arguments (**consider them afterward as undefined**) thus deleting their input.

Exceptions are the copying routines `p_Copy, p_Head`

Most public routines are prefixed by `p_` and require the ring description as last argument (of type `ring`). Some routines do not required a ring, they are prefixed by `p` (`pGetCoeff,pNext,pIter`,..) For historical reasons most routines have also a version with the prefix `p`: they use `currRing` as description of the ring.

6.2 Buckets

An alternative polynomial representation is given by `kBucket7sBucket`. Use `kBucket` if access to the leading term is needed (for example during standard basis computations), otherwise `sBucket` is better suited (for example in sorting polynomials, conversion routines and multiplication).

6.3 CanonicalForm

Another alternative polynomial representation is given by factory's `CanonicalForm`, which represents polynomials in a recursive way, which is well suited for multiplication/division and gcd/factorization routines.

Chapter 7

Interfacing the interpreter

7.1 leftv

The Singular interpreter is stack based, generated by bison. The stack elements are from an array of `sleftv`, therefore `leftv` (pointer to `sleftv`) is the standard type to pass to all the routines. It also explains that the "destructor" `CleanUp` does not free the memory block. The main parts of `sleftv` are the type id and a data pointer:

- `Typ()` the type of the expression (after evaluation): may be the value of rtyp (in the case of a constant) or the type of a variable (then rtyp is `IDHDL`) or the type of an index entry in a list/ideal/etc. (then `e` ist not `NULL`)
- `rtyp` (see `Typ()`)
- `Data()` a read-only pointer to the data of the expression. Has to be casted according to the result of `Typ`
- `data` (see `Data()`)
- `CopyD()` a copy of the data, may only be called once, is more efficient than copy(Data())

7.2 Kernel commands

Kernel commands are scheduled by 4 tables (for 1, 2, 3 or "many" arguments). Which one is used depends on the type of the operation (`CMD_1, CMD_12,..,CMD_M`) and the current number of arguments: for example an operation of type `CMD_12` and 2 arguments would use `dArith2`, while an operation of type `CMD_M` and 2 arguments uses table `dArithM`. The entries for certain operations have to be grouped together, the order within such a block is determined by the order of type conversions: the interpreter tries first to find an exact match for the argument types, after that the first entry where all arguments may be converted is used. For example, for `+` the entry with (matrix,matrix) has to come after (poly,poly): other wise the result of `x+1` would be a matrix and not a poly.

The order of the block is not so important, but an alpabetical order makes them easier to find.

7.3 Procedure calls

Add C/C++ functions via `iiAddCproc("<component>","<name>",<static ?>,<C+` routine to call¿);+ to the current name space. Here `component` stands for the module name (in the file system), used for displaying information, `name` is the name of the function for the interpeter, `static` is 0 or 1.

An alternative may be the procedure from Singular/HOWTO.addKernelCmds, but adding new kernel commands as reserved names should be used only rarely (because of name conflicts, growing table sizes, etc.).

Calling functions (which are not kernel commands) is done via
`iiMake_proc(idhdl pn, package pack, sleftv* sl)`
where pn stands for the function, pack the current package and sl (which may be NULL) the list of the arguments.

7.4 Error handling

All interpreter routines return an error code (FALSE for success, TRUE for error). Additional an explaination should be given via `Werror/WerrorS`. This sets `errorreported,inerror` which the interpreter resets to 0 during error handling. External callers to the Singular interpreter routines have do this themselves.

Chapter 8

Where to find files

8.1 Paths via environment variables and relative to the binary

Singular has 3 mechanism to find its files during run-time:

1. by environment variables
2. relative to the location of the main binary
3. fixed path at configure time

The most important are:

environment var	use	configured place
SINGULARPATH	dirs for libraries	prefix/share/singular/LIB/
SINGULAR_PROCS_DIR	dirs for modules	prefix/lib/singular/MOD/ prefix/libexec/singular/MOD
SINGULAR_INFO_FILE	singular.hlp	prefix/info/singular.hlp
SINGULAR_IDX_FILE	singular.idx	prefix/doc/singular.idx
SINGULAR_HTML_DIR	dir for html files	prefix/singular/html/
SINGULAR_URL	url of the manual	http://www.singular.uni-kl.de/Manual/
SINGULAR_EMACS_DIR	ESingular files	prefix/emacs/

For the configured place `prefix` is set by configure (usually `/usr/local` or `/usr`), for the run-time location `prefix` is `%b/..` where `%b` is the directory of the main Singular binary (after following symbolic links).

For files, the first possibility is chosen, for paths, all existent directories are considered: `system("with");` or `Singular -v` tells the current settings.

8.2 More files

`.singularrc` is searched in the current directory, in the home directory and in the directories of the singular path. If more than one exists, only the first is used.

`gftables/*` is searched in the directories of the singular path and defaults to `prefix/share/factory/gftables/...`

Chapter 9

More environment variables

9.1 SINGULARHIST

If Singular is compiled with `readline`, and if the environment variable `SINGULARHIST` is set and has a name of a valid file as value, then the input history is stored across sessions using this file. Otherwise, i.e., if it is not set, then the history of the last inputs is only available for previous commands of the current session.

9.2 PATH

For calls to external programs, `PATH` is extended by the directory of the main Singular binary and the directories for modules (see previous chapter)

9.3 HOME

tilde in file names is replaced by the contents of HOME. If HOME is not set or unusable, SINGHOME is used (on windos).

Contents