UNIVERSITÄT KAISERSLAUTERN
Zentrum für Computeralgebra

REPORTS ON COMPUTER ALGEBRA
NO. 32

Dynamic Modules in SINGULAR

by

A. Frühbis-Krüger, K. Krüger, H. Schönemann

December, 2003

The Zentrum für Computeralgebra (Centre for Computer Algebra) at the University of Kaiserslautern was founded in June 1993 by the Ministerium für Wissenschaft und Weiterbildung in Rheinland-Pfalz (Ministry of Science and Education of the state of Rheinland-Pfalz). The centre is a scientific institution of the departments of **Mathematics, Computer Science, and Electrical Engineering** at the University of Kaiserslautern.

The goals of the centre are to advance and to support the use of Computer Algebra in industry, research, and teaching. More concrete goals of the centre include

- the development, integration, and use of software for Computer Algebra
- the development of curricula in Computer Algebra under special consideration of interdisciplinary aspects
- the realisation of seminars about Computer Algebra
- the cooperation with other centres and institutions which have similar goals

The present coordinator of the Reports on Computer Algebra is:
Christoph Lossen (email: `lossen@mathematik.uni-kl.de`)

Zentrum für Computeralgebra

c/o Prof. Dr. G.-M. Greuel, FB Mathematik
Erwin-Schrödinger-Strasse
D-67663 Kaiserslautern; Germany
Phone: 49 - 631/205-2850   Fax: 49 - 631/205-5052
email: `greuel@mathematik.uni-kl.de`

Dynamic Modules in Singular

Anne Frühbis-Krüger
FB Mathematatik, Universität Kaiserslautern,67653 Kaiserslautern
Kai Krüger
Fraunhofer ITWM, Erwin-Schrödinger-Str.49,67663 Kaiserslautern
Hans Schönemann
FB Mathematik, Universität Kaiserslautern,67653 Kaiserslautern

December 18, 2003

1 Introduction

When using a computer algebra system, the answer to a particular problem – that is the result of the corresponding calculation – is usually not obtained by just one command but by a (possibly long) sequence of commands. If the method used in this computation is also well adapted to a number of similar problems, this sequence of commands should be automated in some way. In the case of `Singular`, this can be done using the concept of `Singular` libraries (see [5]). But it turns out that this solution is not powerful enough in certain situations, for example if speed is important or if data is available inside the system, but not accessible to the user. For these cases, `Singular` version 2.2 now provides a new mechanism for adding functionality to the system: dynamic modules.

In contrast to a library, a dynamic module is not written in the language `Singular`; it is written in C or C++. Therefore a procedure in a module is not parsed each time the corresponding command is executed, but just once at compile time. This provides a great improvement in speed, if the procedure contains loops with a large number of commands to be executed. Moreover, in a dynamic module, standard C/C++ functions and internal functions of the `Singular` kernel may be used.[1]

On the other hand, writing a dynamic module is not as simple as writing a library procedure, because it does not only require knowledge of C/C++, but also about the way the `Singular` kernel works. In principle, this knowledge

[1]Dynamic modules also provide another advantage - even to those users, who do not write modules themselves: By moving some functions from the `Singular` kernel into modules, the development team of `Singular` can extend the functionality without increasing the size of the kernel. Of course, moving a command to a module only makes sense if this command is not central to a large number of `Singular` applications, for example moving `eigenvalue` to a module is useful, but moving `std` is not.

can be acquired by reading the source code of `Singular`, but this is a very inefficient way. A more efficient way is to leave all the standard parts of the code (for example type-checking) to automated code generation by the module generator `modgen`, a tool for developing `Singular` modules. Of course, writing a useful dynamic module still requires some knowledge about the `Singular` kernel, but usually it is sufficient to know what predefined types are available and what the syntax and output of a few important internal functions should look like.

Section 2 provides an introduction to writing dynamic modules using `modgen`. A detailed explanation of the structure of a module declaration file and the use of the module generator is followed by some comments on the most common problems in the development of a dynamic module and on debugging. In section 3, the technical background of dynamic loading in `Singular` is outlined. Section 4 gives an overview of the structure and philosophy of the `Singular` source distribution and thus provides the means for efficiently extracting additional information directly from the source code.

2 Using the Module Generator

The development of a dynamic module using the module generator consists of three main steps. The first one is writing the module declaration file which contains the C/C++ functions and some general information such as the name of the dynamic module and the help text of it. As the second step, `modgen` generates a directory containing '.cc' and '.h' files for the module as well as a Makefile from the module declaration file. In the last step, the dynamic module can then be built by simply typing 'make' in this directory (possibly after making appropriate changes to the Makefile); it is a shared object file with the appropriate extension for a shared library, e.g. '.so' on Linux- and Solaris-platforms.

2.1 Structure of a Module Declaration File

To define a module, only one file is necessary, which will be referred to as the module declaration file. In this file it is, of course, possible to specify further files which should be used for building the module.
The module declaration file itself consists of 5 main sections:

- the header section, which should at least contain a comment stating the purpose of the module,
- the general declaration section containing the name of the dynamic module, the version and the general help text,
- the SINGULAR section, which may be left out completely, if the the package does not contain any SINGULAR-library procedures,
- the C/C++-section containing the users accessible functionality, and

- the section containing the internal functions, which may be empty, but may not be omitted.

In each of the sections, the usual C- and C++-comments are allowed.

Header Section

This section is the first in the declaration file. Its start is marked by the sequence of characters `%{`, its end by `%}`. Generally speaking, it contains all the lines of C/C++-code, which are supposed to be placed 'as is' in the top part of the main source file of the module. If you need to include system header files or your own header file(s) for the module, the `#include` statement(s) should also be placed here. (Please do not use the name `<module-name>.h` for one of your own header files, as it will be **overwritten** by the module generator.)

Note that you do not have to include the following list of header files explicitly in this section, because they are always included by the module generator:

- system-includes from /usr/include:

stdlib.h	stdio.h	string.h
ctype.h	unistd.h	sys/stat.h

- Singular-includes:

mod2.h	tok.h	structs.h
ipid.h	locals.h	omalloc.h

- header file of the module.

Example:

```
%{
/*
 *  $Id: kstd.mod,v 1.1 2002/05/10 15:20:07 anne Exp $
 *
 *  Module file for computing standard bases only taking into
 *  account the first components of a module
 *
 */

#include <ideals.h>
#include <ring.h>
#include <kstd1.h>
#include <prCopy.h>
%}
```

General Declaration Section

This section contains the general information about the dynamic module, using the following syntax:

```
package     =  ''<module-name>''
version     =  ''<version>''
info        =  ''<help for the module>''
[category   =  ''<category>'']
[files      =  <cxxsource1[,cxxsource2...]>]
```

The use of the keywords `version`[2], `info` and `category` is exactly the same as in a Singular Library (see [5]), the new keyword `package` specifies the name of the dynamic module. The optional keyword `files` allows further C- and C++-files to be specified; these are assumed to be in the same directory as the module declaration file and will be compiled and used for linking in the process of building the dynamic module.

Example:

```
category="internal";
package="kstd";
version="$Id: kstd.mod,v 1.1 2002/05/10 15:20:07 anne Exp $";
info="
LIBRARY: kstd.lib    Standard Bases on First Components only

PROCEDURES:
   kstd(module i1, int i2);    std on first i2 components of i1 only;
";
```

In addition to these variables, the section also provides the possibility to specify a block of code to be executed when the module is loaded. Usually this is only necessary, if the module relies on other dynamically loaded parts of SINGULAR. The block of code, which can be placed at any point of this section, should start with the keyword `%modinitial` and end with `%endinitial`.[3]

SINGULAR **Library Section**

The start of this section is marked by the tag `%Singular`; the syntax is exactly the one of a SINGULAR library with the exception that the keywords `package`, `version` and `info` are not allowed. Moreover, exiting from the procedure by any other way than a `return` or an `ERROR` statement leads to an error message about a misplaced }.

By using the module generator, this section is transformed into a special type

[2]For the recommended syntax of the version string, please refer to the CVS manual [1].

[3]If such a block is not present in the module declaration file, these two keywords may be left out completely.

of Singular library (with name <module-name>`.bin`), which is loaded automatically when loading the dynamic module.
The purpose of this section, is to provide simultaneous access to routines from the dynamic module and procedures written in the programming language of Singular within the same package.
Warning: Editing a `.bin` file directly, will make it unusable, because the dynamic module relies on the procedures being exactly in the expected positions in the file and checks it by crc-checksums.

User Accessible Commands of the Module

The tag `%procedures` marks the beginning of the section which provides those user accessible commands of the module which are written in C/C++.

Although this section contains C/C++ sources, there are certain simplifications:

1. All classes and all internal routines of Singular are available (provided the correct header files have been included).
2. Types appearing in the declaration line of a command are the same as the ones in the declaration line of a Singular library routine.
3. Declaration and typechecking of objects handed to the procedure as parameters may be substituted by the keywords `%declaration` and `%typecheck`. The `modgen` recognizes these keywords and replaces them by the approriate code.
4. Returning objects at the end of the procedure needs to be done by the keyword `%return`. The right-hand-side of a `%return` statement should contain an object to be handed back to the user, not some internal data structure.

Example:

```
%procedures

module kstd (module h1, int k)
{
// automatic declaration of h1 and k
%declaration;

// variables used in this procedure
  ideal s_h1;
  ideal s_h3;
  int j;
  ring orig_ring;
  ring syz_ring;
  intvec *w=NULL;
```

```
// automatic typechecking of h1 and k
%typecheck;

// use of internal SINGULAR routines
  assume(currRing != NULL);
  orig_ring=currRing;
  syz_ring=rCurrRingAssure_SyzComp();
.
.
.
// returning the result s_h3
  %return = (void *)s_h3;
}
```

Internal Routines

The last section only contains routines which are not directly accessible from the user interface of SINGULAR. As before objects and routines from the SINGULAR kernel are available, but here the keywords `%declaration`, `%typecheck` and `%return` are not allowed, because the content of this section is copied identically to the `.cc` file.

2.2 Using the module generator `modgen`

Taking the name of the module declaration file as an argument, the utility program `modgen` (part of the SINGULAR distribution) generates the necessary files for building the corresponding dynamic module. More precisely, it's default behaviour is the following: It creates a directory whose name is the basename of the module declaration file and which contains five files.

<basename>.cc	C/C++-source code automatically generated from the header sections and the %procedures and %C sections
<basename>.h	automatically generated main header file of the module
<basename>.bin	the contents of the %Singular section in in a special pre-parsed format
<basename>.pl	the help sections of the user accessible command from the %procedures and %Singular sections in a special pre-parsed help format
Makefile	Makefile for the dynamic module containing all further information necessary to build the module. In particular, important configuration switches like CXX and CFLAGS inherited from the sources of the SINGULAR kernel.

This default behaviour of `modgen` can be altered by command-line options in the following way:

Synopsis:
`modgen [-dhmsv] [-i <install-dir>]` <file-name>

Options:

-d	print debugging information
-h	print syntax and short descriptions of the options
-m	turn off the creation of the Makefile[4]
-s	create the files in the current directory instead of a subdirectory[5]
-v	verbose mode
-i <install-dir>	specify the destination directory for `make install`

2.3 Avoiding Name Conflicts: Namespaces

As more and more people are writing functions and libraries for SINGULAR, the risc of different authors giving the same name to their function is increasing. If the function or library is written in Singular, the function can be renamed without too much effort, but if the function is a C/C++ function compiled into a binary library the only way is to rename the function in the source file and recompile the library (of course, for each operating system separately). To avoid such conflicts, the concept of namespaces has been introduced into SINGULAR. The basic idea is to split the set of available identifiers in SINGULAR into subsets.

To understand this concept, it is necessary to understand the way SINGULAR handles its identifiers; in this context, identifiers are all items which are neither an internal function nor a reserved word. SINGULAR stores these items in a single list, which is organized in a very special way: Identifiers corresponding to ring-independent objects are stored directly in this list, those corresponding to ring-dependent objects are stored in a sublist which is connected to the main list at the identifier corresponding to the ring. The concept of a namespace extends this idea of sublists by introducing a new type `package` which allows the use of a sublist not only for ring-dependent objects but for any identifier (except for identifiers corresponding to a package). Such a namespace is automatically created, whenever a library is loaded; it then contains all procedures from that library and in addition to that all variables created as local variables in by procedures from the library. On the other hand, a namespace can also be created interactively by the user by defining a variable of type `package`.

[4]By using this switch changes to the Makefile will not be overwritten the next time when `modgen` is being used.

[5]If there is a large number of additional C/C++-source files belonging to the dynamic module, it is preferable to put these files and the module declaration file into the same directory and have `modgen` create the file there.

So each library corresponds to a subset of the set of all valid identifiers. Within each subset, the author of the library has to take care of avoiding name-conflicts. To access to such an identifier, its complete name including the name of the package `<subset-name>::<identifier>` has to be specified.

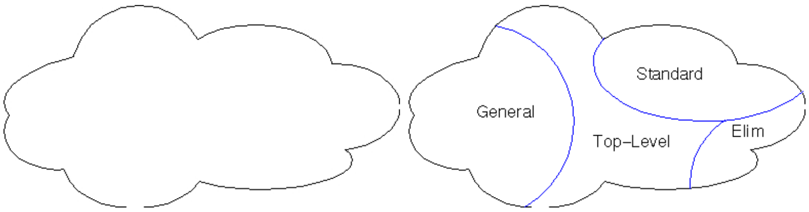

Figure 1: Left: structure of the set of all names after loading the libraries `standard.lib`, `elim.lib` and `general.lib` in SINGULAR version < 2.2; Right: the same in SINGULAR version ≥ 2.2

When SINGULAR is started, there is only one namespace, called `Top`. This is the namespace in which all variables reside which were declared in interactive use. For reasons of convenience and backward compatibility, it is possible to omit `Top::` when referring to objects of this namespace. Whenever a new library or dynamic module is loaded, a namespace is added; the name of this namespace is automatically generated from the name of the library or module by taking the basename of it and converting the first letter to upper case. The spelling of the remaining part of the basename is unchanged.

At the SINGULAR user interface all namespaces appear as variables of type `package`; their contents can be viewed by `listvar(<package-name>)`. New namespaces can also be created directly by defining a variable of type `package`, variables[6] can be introduced to it by just prepending `<subset-name>::` to the name in it the definition of the variable, and namespaces can be removed like usual variables by the command `kill`.[7]

Using namespaces, the only remaining scenarios of name conflicts are:

- Two libraries have the same name.
  Solution: Rename one of the two.[8]

- Two libraries contain commands of the same name and are being loaded by the command `LIB`.

[6] In this context a `proc` is considered a variable, too.

[7] The traditional object in SINGULAR which is closest to the new type `package` is the type `ring`; both types are designed to contain objects of various other types and extra information like info- and version-strings in the case of packages and characteristic and ordering in the case of rings.

[8] Renaming libraries originating from the SINGULAR distribution leads to non-standard identifier names for the procedures from these libraries. This may cause errors in other libraries which are part of the distribution.

Solution: Load both of them by the command `load()` and prepend the appropriate prefix to all commands from the namespaces when referring to them.

2.4 Some Remarks on Writing Libraries and Dynamic Modules

After discussing the technical aspects of writing a dynamic module (using the module generator) and the basic concept of namespaces, another important issue is the decision whether writing a dynamic module is appropriate for a specific task or whether a SINGULAR library is more suitable. The main criteria in favor of writing a module are speed and full access to internal structures and routines; those in favor of a library procedure are platform-independence of the resulting library file and the convenience of writing in the same language as in the user interface. In more detail, the most important differences are:

SINGULAR Library	Dynamic Module
The syntax coincides with the syntax of the interactive user environment of SINGULAR.	The source files should be written in the programming language C++ (possibly with part in C), making use of the class definitions in the SINGULAR kernel.
SINGULAR libraries are plain text files and hence independent of the platform SINGULAR is used on. They are interpreted at run-time.	Dynamic modules are binary files which have been compiled and linked for a specific platform (i.e. hardware and operating system).
Any line of commands is interpreted each time it is encountered. Therefore lines inside loops are treated more than once (which is, of course, time consuming).	The source code is compiled once.
If SINGULAR is upgraded to a new version, usually no changes to the library are necessary.	For ensuring that the module still works properly after upgrading SINGULAR, it should at least be recompiled.
Syntax errors (except bracket mismatches and similar structural mistakes) are not detected before the corresponding line of code is executed.	Syntax checks are done at compile time for all lines of code.

Commands from other SINGULAR libraries are accessible, but no internal routines or data structures of the SINGULAR kernel unless they correspond to a command in the user interface.	Internal routines and data structures of the SINGULAR kernel are accessible, but calling a SINGULAR library procedure is difficult.
Debugging of procedures can be done using the appropriate tools of the SINGULAR interpreter.	For debugging, external tools, like e.g. gdb, may be necessary.

In practice, the decision between a library and a dynamic module is usually not obvious; in most cases, there will be good reasons for implementing certain parts in C/C++ and others in a SINGULAR procedure. This is possible by putting the corresponding parts into seperate routines and will not lead to any problems as long as no command from the `%procedures` section calls a command from the `%Singular` section. This leads to a combined module which is loaded by loading the shared library file - the corresponding `.bin` file is then read automatically.[9]

When writing dynamic modules, another issue which should not be forgotten is error handling. In SINGULAR the standard way of reacting to an error is printing out an error message using the function `Werror` and returning the boolean value TRUE.[10] The syntax of `Werror` is the syntax of `printf`, but it also sets the global variable `errorreported` to announce that the error message for the error which just occurred has already been printed. For generating warning messages, another very similar function is available, `Warn`, which also has the same syntax as `printf`. In general, the C function `printf` should never be used in a dynamic module, because its output can, e.g., not be saved into the session protocol by using the SINGULAR command `monitor`. Whenever `Werror` and `Warn` are not appropriate, but output should be generated, `printf` should be replaced by `Print`, which can be used in exactly the same way.

2.5 Debugging

Since a dynamic module is a shared library, in particular a binary file, simple syntactical errors are already detected when compiling. But most errors are of a more subtle nature and require debugging. Unfortunately, a dynamic module is not a program by itself and can hence not be run directly in a debugger; thus a debuggable binary of SINGULAR, which in turn loads the dynamic module, has to be run in a debugger.

[9] When loading the `.bin` file, a checksum test makes sure that the version of the file corresponds to one of the shared library

[10] This should be the behaviour in all situations where the SINGULAR command ERROR would be used when writing a library.

There are several kinds of debuggable binaries of SINGULAR:

`Singulara` Debuggable version of SINGULAR which is closest to the usual binary: compiled without option `-fomit-frame-pointer`, with memory testing functions and with optimization activated

`Singularb` Profiling version of SINGULAR: prepared for basic block profiling, optimization activated (`-O -g`), without additional tests, requires bprof

`Singularg` Debuggable version of SINGULAR: compiled without optimization, with option `-g` and with the full set of additional tests activated

`Singularp` Profiling version of SINGULAR: optimization activated (`-O -g`), without additional tests

`Singulart` Specialized debugging version for finding memory leaks: keeping records for each memory block about where/when it was allocated, where it was freed etc.; otherwise similar to `Singulara`

For debugging a dynamic module, `Singulara` is usually the most suitable debuggable binary, because it does not contain too many extra testing functions, is still optimized and is hence not very slow. If it turns out during debugging that `Singulara` does not provide all the necessary information, `Singularg` should be chosen.

3 The Principle of Dynamic Loading in SINGULAR

To understand how dynamic loading is implemented in SINGULAR, it is necessary to understand the principles of shared libraries and to consider the internal structure of SINGULAR. For writing dynamic modules, the information in this section is technical background information which can safely be skipped.

3.1 Accessing Functions - General Situation

At compile time, a symbol is generated for each function which is defined in the source: for functions originating from C sources, the symbol conincides with the name of the function, whereas the symbol of a C++ function additionally encodes the return type and the parameter types. At link time, the table of symbols is generated, which contains the symbols and the addresses at which the corresponding functions can be retrieved, that is the destination of a jump to the entry point of the given function; the symbol table is then stored at a specific location in the file header. The address corresponding to a symbol is usually fixed within the executable file. For use in a shared library which is loaded at run time, however, the address of the function cannot be an absolute address inside an executable file, since the code of the library is not included into program binaries and programs accessing a library function should still work properly after the library has been recompiled/relinked. This dilemma is resolved by only adding information on predefined entry points to the library into the binary file, putting the relative address information for the library functions into a table inside the library binary and keeping all address information inside a function relative to the beginning of the function, not the one of the library. Code, which is to be used inside a shared library and thus needs relative addressing, has to be compiled from source as position independent code (PIC).

Using a shared library generated in the above way, an executable can now call library functions by means of the predefined entry points. But in our context, the shared libraries, that is the dynamic modules, are designed to extend the functionality of SINGULAR. Therefore it is also necessary that the library functions are able to access symbols of the SINGULAR kernel. This functionality cannot be emulated in an efficient and consistent way for a program with as many symbols as SINGULAR (approx.8500) and hence must be provided by the underlying operating system where possible. Linux, SunOS/Solaris and many of the other commonly used Unix-like operating systems support this feature, but Microsoft's operating systems, for instance, do not. In the second case, we are forced to emulate this by means of a tranlation table; as a full translation table would be huge and would need constant updating, the one we are using only contains a relatively small subset of the symbols of the SINGULAR kernel.

3.2 Accessing Functions in a Dynamic Module

When the program SINGULAR is started, no information about procedures in SINGULAR libraries and about functions in dynamic modules is present, until a library or module is loaded. Upon loading a library, all information abount the procedures in the library can be determined by parsing the ASCII-file containing the library; but a dynamic module is a binary file. Therefore, there has to be a well-defined entry point to the dynamic module which, in turn, passes all the necessary information on to SINGULAR. This entry point is the function `mod_init` whose symbol is again `mod_init`, because the function is written in C. The argument to this function call is the pointer to the specialized SINGULAR kernel function `iiAddCproc` for adding new functions. This function is then used to add identifiers corresponding to the functions defined in the dynamic module to the list of identifiers in SINGULAR[11].

More precisely, when loading a dynamic module, say `dummy.so`, SINGULAR performs the following steps:

1. Generation of a new namespace `Dummy`
   The name of the namespace is derived from the name of the dynamic module by converting the first letter to upper case and all the following ones to lower case.

2. Change of current namespace to `Dummy`

[11] At this point the functions will become visible to the user, for instance, in the output of the command `listvar(<Packagename>);`

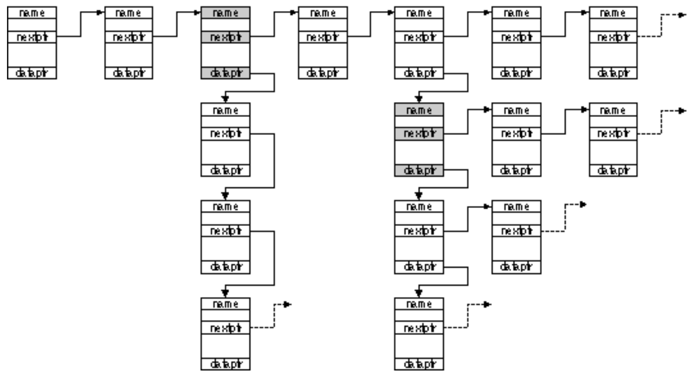

Figure 2: The internal identifier list

3. Loading of the binaries of the module
   To this end, the generic loading function `dynl_open` is used, which masks the actual loading function provided by the operating system (e.g. dlopen for Linux, shl_load for HP-UX etc.).

4. Determining the address of the entry function `mod_init`
   For this task the function `dynl_sym` is provided. The name of the entry function is predefined ensuring that the corresponding symbol is present. If the module is generated by `modgen`, the function `mod_init` has been generated automatically.

5. Call of the procedure `mod_init`
   As parameter `iiAddCproc()`, the SINGULAR-kernel routine for adding new elements to the identifier list is specified. `mod_init` then registers all functions, which are provided by the module, and their addresses to the identifier list.

6. Change of current namespace to the original one. The module is now loaded.

In the above explanation of loading a dynamic module, creation of and access to a namespace is mentioned. These notions can only be understood with the structure of the internal identifier list in mind. All identifiers are stored in a list where identifiers of type ring and type package are the branching points. Creation of a namespace refers to adding a new identifier of type package to the list and thus opening a new branch; accessing a namespace means that all search, insert and delete operations will be performed in the sublist belonging to the package, not in the main list. Figure 2 illustrates the structure of the identifier list.

4 Anatomy of Singular

For writing dynamic modules, which extend the functionality of Singular in a useful way, it is necessary to have some knowledge about the kernel of Singular. Singular's source code is publicly available under the Gnu Public License (GPL). It is therefore unnecessary to describe all details of the program explicitly - which would, by the way, take several hundred pages. Instead, we provide a brief introduction to the general structure and to the philosophy of the way the sources are organized, hoping that this will enable users to efficiently search and find the information they need.

4.1 Directory Layout

Singular uses several specialized libraries[12], whose source code can be found in the corresponding directories of the source tree; further directories are devoted to the Singular kernel itself, to the module generator and to specialized functionality such as non-commutative computations and integer programming.

To the majority of the directories of the source tree, the Gnu Public License applies, but several other directories are subject to different licensing conditions which are compatible with the GPL. Details on the author and the license of a specific directory are available in the file `LICENSE` in the respective directory.

In the following list, the content of each of the subdirectories is explained:

- `omalloc`
  memory manager specialized for the need of `Singular`, used in all parts of Singular
- `gmp` (Gnu Multiple Precision Library – GMP [2])
  GNU MP is a library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers.
  This library provides the arithmetic on long integers to Singular and to Singular–Factory.
- `ntl`
  NTL – A library for doing numbery theory [3].
  It provides univariate factorization to Singular–Factory.
- `factory` Singular-Factory
  Factory is a C++ class library based on a recursive representation of multivariate polynomial data.
  It provides factorization, GCD etc. to Singular.
- `libfac`
  Extension to Singular–Factory, providing charactistic sets and multivariate factorization in charactistic p.

[12]Singular needs the standard C–libraries `libc`, `libm`, `libdl` and `libncurses` (resp. `libtermcap`); for compiling with gcc 3.x, Singular additionally needs `libstd++`.

- MP
  Multiprotocol – the communication protocol for exchange of mathematical data via tcp-connections and via compact binary files [4]

- Singular
  SINGULAR kernel

- IntegerProgramming
  Several stand alone programs for use with SINGULAR: Procedures for Toric ideals/Integer Programming using Groebner basis methods. Its interface to Singular can be found in intprog.lib and toric.lib.[13]

- modules
  Module generator, supporting files and the provided module sources

- Plural
  Non-commutative functionality of SINGULAR

- doc
  TeXinfo documention of SINGULAR

- Tst
  automated tests of SINGULAR

4.2 Interpreter types and C++-types

The available types on the level of SINGULAR's user interface (i.e. on the interpreter level) and functions related to them are documented in the user manual. In the backend of SINGULAR (i.e. in the C/C++ code), each of these interpreter types has its equivalent; functions dealing with a specific type can be identified by the prefix of their name. In the following table, these corrrespondences are listed:

Correspondence of SINGULAR types to C/C++ types		
SINGULAR	C/C++	function prefix
int	int	-
string	char *	-
intvec	intvec *	iv
intmat	intvec *	iv
number	number	n, np, nl, na, n_
poly	poly	p, p_
vector	poly	p, p_
ideal	ideal	id
module	ideal	id
matrix	matrix, sometimes also ideal	mp

[13]For the mathematical background see section *M*athematical Background in the SINGULAR manual.

`resolution`	`resolution`	-
`list`	`lists`	l
`ring`	`ring`	r
`qring`	`ring`	r
interpreter related types		
`package`	`package`	
expression, expression list	`leftv`	

The general rule for generating names of basic functions is `<prefix><Name>`, where the first letter of the main part is capitalized. This rule applies, in particular, to the arithmetic for each type; the addition, for instance, is provided by the following functions:

- `pAdd(a,b)` - addition of the polynomials resp. vectors `a` and `b` in the current ring
- `nAdd(a,b)` - addition of the numbers `a` and `b` in the current ring
- `idAdd(a,b)` - addition of the ideals resp. modules `a` and `b` in the current ring
- `mpAdd(a,b)` - addition of the matrices `a` and `b` in the current ring
- `ivAdd(a,b)` - addition of the integer vectors resp. integer matrices
- `p_Add(a,b,r)` - addition of the polynomials resp. vectors `a` and `b` in the ring `r`
- `n_Add(a,b,r)` - addition of the numbers `a` and `b` in the ring `r`

4.3 The SINGULAR interpreter - link between user commands and C/C++ functions

The names of more advanced functions often do not follow the general building rule which was outlined above, but rather encode their general purpose in the name. Moreover, SINGULAR is a system which has grown over more than two decades, so the naming of some functions is sometimes even a bit misleading. All this makes searching for functions by name a very tedious task; fortunately, this can be avoided in most situations by starting the search on the interpreter level. To be able to efficiently search in this way, it is important to keep some details about the interpreter in mind.

The user input is parsed by a scanner/parser combination which is defined by the files `scanner.l` and `grammar.y` and is usually evaluated immediatly. The basic interpreter type is a `leftv` which describes an expression: it has a (possibly empty) name, a type and a value. The most important methods of the `leftv` class are `Typ()` which returns the type of an expression and `Data()` resp. `CopyD()` which returns its value as read-only data resp. a copy thereof. Operations with objects of type `leftv` are performed in the routines `iiExprArith1`, `iiExprArith2`, `iiExprArith3` or `iiExprArithM` respectivly - depending on the number of operands: one, two, three or a variable number. These routines dispatch the work to others via the tables `dArith1` to `dArithM`, and handle the return types, and the error conditions (a return value of TRUE shows an error).

These operations and tables are defined in the file `iparith.cc`, which is not really accessible to human reading. A more human readable form of these tables appears as a comment in the file `iparith.inc` which is automatically generated at compile time on the basis of `iparith.cc`.

Therefore, a good approach to searching for a function which provides a specific functionality is the following: first choose a SINGULAR command which probably relies on this functionality, look it up in the tables in `iparith.inc`, find the name of the function to which the task is dispatched and then search for this function in the source code.

Interpreter related functions usually return an error condition as function value and pass the computed result in their first argument. Therefore it is not suitable to pass an error message via an error condition in the usual way; instead, an error message should be reported via the functions `Werror` or `WerrorS` precisely in the routine where it occurs. All functions should return TRUE upon failure of the function itself or of one of its subroutines. The routines `iiExprArith1` etc. print a generic error message if none was generated using a `Werror/WerrorS` call up to that point.
To illustrate this process of descending from the interpreted language to the C/C++ functions, we consider the example of the addition of two numbers which is syntactically an expression on the level of the user interface, say

```
a + b;
```

The interpreter calls

```
expr '+' expr { if(iiExprArith2(&$$,&$1,'+',&$3)) YYERROR; }
```

`iiExprArith2` dispatches this call to the function

```
static BOOLEAN jjPLUS_N(leftv res, leftv u, leftv v)
```

which, in turn, calls the "real" function:

```
res->data = (void *)(nAdd((number)u->Data(), (number)v->Data()));
```

4.4 Important files in the Singular directory

Although it is possible to find all necessary information using the search process described in the last section, it is often convenient to know the content and purpose of the most central files of the SINGULAR kernel. In particular it is always useful to know where to look for general definitions and for the operations on the basic types. Table 3 gives an overview of the most important header files. On the other hand, some files are automatically generated at compile time which implies, in particular, that all changes to these files will inevitably disappear (at the latest) the next time `make clean` is called; these files and the corresponding original files are listed in table 3, as well.

some important files	
global definitions	
`mod2.h`	generated by `configure` contains all general configurable parameters
`structs.h`	general structures/classes/types/macros
`tok.h`	enumerations and macros, general global variables for the interpreter
`grammar.h`	generated by bison from grammar.y: the generated enumerations
operation with basic types	
`numbers.h`	operations of the (general) coefficient type `number`
`modulop.h`	operations of the coefficient type `number` in Z/p
`longrat.h`	operations of the coefficient type `number` in Z resp. Q
`longalg.h`	operations of the coefficient type `number` in algebraic/trancendental extensions
`polys.h`, `p_polys.h`, `polys-impl.h`	operations of the type poly
`ideals.h`	operations of the type ideal
`matpol.h`	operations of the type matrix (of polynomials)
`intvec.h`	operations of the type intvec
standard bases	
`kstd1.h`	local orderings: algorithm of Mora
`kstd2.h`	global orderings: algorithm of Buchberger
`syz.h`	syzygies (see also `ideals.h`)
interpreter related	
`ipid.h`	identifier handling
`ipconv.h`	type conversions
`ipprint.h`	formatting output
`ipshell.h`	miscellaneous utilities for the interpreter
automatically generated source files	
`mod2.h`	generated by `configure`
`grammar.h`	generated by `bison` from `grammar.y`
`grammar.cc`	generated by `bison` from `grammar.y`
`scanner.cc`	generated by `flex` from `scanner.l`
`libparse.cc`	generated by `flex` from `libparse.l`
`version.h`	generated by make (contains the date)
`static.h`	generated by make (contains `HAVE_STATIC`)
`prCopy.inc`	generated by `perl` from `prCopy.pl`

p_Procs_Static.inc	generated by p_Procs_Static from p_Procs_Generate.cc
p_Procs_Dynamic.inc	generated by p_Procs_Dynamic from p_Procs_Generate.cc
iparith.inc	generated by gentable from iparith.cc
mpsr_Tok.inc	generated by gentable from iparith.cc
feOpt*.inc	generated by feOpt from feOpt.cc

References

[1] CVS Documentation,http://www.cvshome.org/docs/ref.html

[2] GNU MultiPrecision library http://www.swox.com/gmp

[3] NTL Number Theory Library http://www.shoup.net/ntl

[4] MP http://www.symbolicnet.org/areas/mp.html

[5] Singular Online documentation http://www.singular.uni-kl.de

Papers published in the Reports on Computer Algebra series

[RCA:32] A. Frühbis-Krüger; K. Krüger and H. Schönemann. Dynamic Modules in Singular. December 2003.

[RCA:31] G.-M. Greuel, C. Lossen and M. Schulze. Three Algorithms in Algebraic Geometry, Coding Theory, and Singularity Theory. March 2001.

[RCA:30] G.-M. Greuel. Applications of Computer Algebra to Algebraic Geometry, Singularity Theory and Symbolic-Numerical Solving. June 2000.

[RCA:29] G.-M. Greuel. Computer Algebra and Algebraic Geometry – Achievements and Perspectives. Feb 2000.

[RCA:28] T. Siebert. Recursive Computation of Free Resolutions and a Generalized Koszul Complex. Jan 2000.

[RCA:27] O. Bachmann and H.-G. Gräbe. The SymbolicData Project: Towards an Electronic Repository of Tools and Data for Benchmarks of Computer Algebra Software. Jan 2000.

[RCA:26] B. Reinert. Solving One-Sided Equations in Integer Monoid Rings. Jan 2000.

[RCA:25] B. Reinert and D. Zeckzer. Coset Enumeration using Prefix Gröbner Bases in Mrc – An Experimental Approach. Nov 1999.

[RCA:23] Birgit Reinert. Observations on coset enumeration. Nov 1998.

[RCA:22] K. Madlener and F. Otto. Some Applications Of Prefix-Rewriting In Monoids, Groups, And Rings. November 1998.

[RCA:21] G.-M. Greuel, G. Pfister, and H. Schönemann. Singular version 1.2 user manual. June 1998.

[RCA:20] B. Reinert and D. Zeckzer. MRC – A System for Computing Gröbner Bases in Monoid and Group Rings. July 1998.

[RCA:19] B. Reinert, K. Madlener, and T. Mora. A note on nielsen reduction and coset enumeration. February 1998.

[RCA:18] O. Bachmann and H. Schönemann. Monomial Representations for Gröbner Bases Computations. January 1998.

[RCA:17] Thomas Siebert. An algorithm for constructing isomorphisms of modules. January 1998.

[RCA:16] K. Madlener and B. Reinert. String Rewriting and Gröbner Bases – A General Approach to Monoid and Group Rings. October 1997.