The BLAD libraries (version 6.0)

François Boulier

Copyright © 2004-2026 François Boulier

Table of Contents

1 Introduction 1

1.1 Introduction (2023) 1
1.1.1 Structure of the libraries 1

2 Configuration and Building 2

The BLAD libraries copying conditions 2
2.1 Building the development version 2
2.2 Building the BLAD libraries 3
2.2.1 Recommended build on a Linux machine 3
2.2.2 Basic build 3
2.2.3 GMP 4
2.2.4 mini-gmp 4
2.2.5 Building example under Darwin 4
2.2.6 Miscellaneous 4
2.2.6.1 Tests for debugging purposes 4
2.2.6.2 Apple Universal OS X 4
2.2.6.3 32 bits and 64 bits versions 4
2.2.7 What gets installed 5
2.2.8 Using the libraries after installation 5
2.2.8.1 A small C program using BLAD 5
2.2.8.2 A small C++ program using BLAD 6
2.2.8.3 Caveats 6
2.2.9 Reporting bugs 6
2.3 Building BLAD under MS-Windows 7

3 The developer corner 8

3.1 The tools directory 8
3.2 Implementing a new library 8
3.2.1 The `src` directory 9
3.2.2 The `doc` directory 9
3.2.3 The `tests` directory 9
3.2.4 Modifications to the other libraries 9
3.2.5 Building the library 10

4 The ba0 library 11

4.1 Introduction 11
4.1.1 Basic types, constants and naming conventions 11
4.1.2 Checking the version number of the library 11
4.1.3 Global variables and settings 11
4.1.4 Sequences of calls to the library 18
4.2 Memory management 20
4.2.1 Overview 20

4.2.2 System memory allocation 20
4.2.3 GMP memory allocation 20
4.2.4 Stacks 21
4.2.4.1 Stack settings 22
4.2.4.2 Memory alignment 22
4.2.4.3 Stack primitives 22
Stack debugging primitives 24
4.2.4.4 Stack of stacks primitives 25
4.2.4.5 Mark primitives 25
4.2.5 Garbage collection 26
4.2.5.1 The basic scheme 26
4.2.5.2 The two stacks scheme 27
4.2.5.3 The garbage function 27
4.3 Containers 30
4.3.1 Overview 30
4.4 Formats 31
4.5 An example of a new data structure definition 33
4.6 Exceptions 37
4.6.1 Basic exceptions 37
4.6.2 Exceptions with parameters 38
4.6.3 Data structures 39
4.6.4 The exception extra stack 40
4.6.5 Exceptions primitives 40
4.7 Input and output 42
4.7.1 Basic output 42
4.7.2 Basic input 44
4.7.3 The lexical analyzer 45
4.7.3.1 The context string 45
4.7.3.2 Lexical analyzer setting 45
4.7.3.3 The substitution dictionary 46
4.7.3.4 Lexical analyzer primitives 46
4.7.4 High level input and output 49
4.8 Generic functions 50
4.9 Lists 51
4.9.1 Generic lists 51
4.9.2 Lists of machine integers 52
4.9.3 Lists of floating point numbers 53
4.10 Tables 53
4.10.1 Generic tables 53
4.10.2 Tables of machine integers 56
4.10.3 Tables of floating point numbers 57
4.11 Matrices 59
4.11.1 Generic matrices 59
4.11.2 Matrices of machine integers 60
4.11.3 Matrices of floating point numbers 61
4.12 Values 62
4.12.1 Generic values 62

4.12.2 Values of machine integers . . . 62
4.12.3 Values of floating point numbers . . . 63
4.13 Points . . . 63
4.13.1 Generic points . . . 63
4.13.2 Points with machine integer values . . . 65
4.13.3 Points with floating point values . . . 66
4.14 Numbers . . . 67
4.14.1 Machine integers . . . 67
4.14.2 Bools . . . 68
4.14.3 Big integers . . . 69
4.14.4 Modular machine integers . . . 70
4.14.5 Modular big numbers . . . 70
4.14.6 Big rational numbers . . . 71
4.14.7 Floating point numbers . . . 72
4.14.8 Intervals with big rational bounds . . . 72
4.15 Strings . . . 78
4.15.1 Plain strings . . . 78
4.15.2 Indexed strings . . . 79
4.15.3 Range indexed groups . . . 82
4.16 Dictionaries . . . 86
4.16.1 Dictionaries with generic pointer keys . . . 86
4.16.2 Dictionaries with machine integer keys . . . 87
4.16.3 Dictionaries with string keys . . . 88
4.16.4 Dictionaries with typed string keys . . . 90

5 The bav library . . . 92

5.1 Introduction . . . 92
5.1.1 Settings . . . 92
5.1.2 Basic types, constants and naming conventions . . . 92
5.2 Global variables and settings . . . 93
5.3 Differential rings . . . 95
5.4 Orderings . . . 98
5.5 Current ordering . . . 103
5.6 Typed identifiers . . . 103
5.7 Blocks . . . 105
5.8 Subrankings . . . 107
5.9 Symbols . . . 108
5.10 Dictionaries with symbol keys . . . 111
5.11 Parameters . . . 113
5.12 Variables . . . 116
5.13 Dictionaries with variable keys . . . 125
5.14 Ranks . . . 126
5.15 Terms . . . 128
5.16 Term orderings . . . 133
5.17 Points . . . 133

6 The bap library 134
6.1 Introduction 134
6.1.1 Global variables and settings 134
6.1.2 Basic types, constants and naming conventions 134
6.2 Internal representation of polynomials 135
6.2.1 Ordered linear combinations of terms 135
6.2.1.1 Zipped terms 135
6.2.1.2 Term managers 135
6.2.1.3 Clots 137
6.2.1.4 Iterators of monomials over a clot 138
6.2.1.5 Creators of clots 140
6.2.2 Polynomials 142
6.2.2.1 Sequential accesses 142
6.2.2.2 Indexed accesses 142
6.2.2.3 Iterators for indexed accesses 143
6.2.2.4 Creators of indexed accesses 144
6.2.2.5 Term strippers 145
6.2.2.6 Polynomials 146
6.3 Primitives for polynomials 146
6.3.1 Miscellaneous low level primitives 146
6.3.2 Initialization and assignment primitives 148
6.3.3 Predicates 150
6.3.4 Comparing and sorting 151
6.3.5 Degree, rank, initial, separant 152
6.3.5.1 Parsing and printing polynomials 157
6.3.5.2 Addition of polynomials 158
6.3.5.3 Multiplication of polynomials 159
6.3.5.4 Division and pseudo division of polynomials 160
6.3.5.5 The subresultant algorithm 162
6.3.5.6 Differentiation and related primitives 164
6.3.5.7 Evaluation of polynomials 165
6.3.5.8 Geobuckets 166
6.3.6 Polynomials with `mpz_t` coefficients 167
6.3.6.1 Primitives applying on numerical coefficients 167
6.3.7 Polynomials with `mpq_t` coefficients 168
6.3.8 Polynomials with modular `mpz_t` coefficients 169
6.3.9 Polynomials with modular machine integer coefficients 170
6.4 Iterators and creators for polynomials 171
6.4.1 Enumerating polynomial coefficients 171
6.4.2 Enumerating and creating numerical coefficients 172
6.4.2.1 Enumerating numerical coefficients 172
6.4.2.2 Creating polynomials 174
6.5 Products 176
6.5.1 Generic products 177
6.5.2 Products with `mpq_t` coefficients 180

7 The baz library 181

7.1 Introduction 181
7.1.1 Global variables and settings 181
7.1.2 Basic types, constants and naming conventions 181
7.2 Primitives 182
7.2.1 The gcd and related methods 182
7.2.2 The factorization and related methods 184
7.2.3 Miscellaneous primitives 185
7.2.4 Primitives applying to products 186
7.2.5 Isolation of real roots 186
7.3 Rational fractions with `mpz_t` coefficients 187
7.3.1 Relational expressions between rational fractions 195
7.4 Prolongation patterns 196
7.5 Series Mappings 198
7.6 Points with rational fraction values 199

8 The bad library 202

8.1 Introduction 202
8.1.1 Basic types, constants and naming conventions 202
8.2 Global variables and settings 203
8.3 Properties of regular chains 204
8.4 Regular chains 206
8.5 Intersections of regular chains 213
8.6 Quenching regular chains 216
8.7 Base fields 220
8.8 Reduction 223
8.9 Regularization 226
8.10 Normal forms 229
8.11 Reduced forms 231
8.12 Critical pairs 231
8.13 Selection strategies 234
8.14 Splitting trees 235
8.15 Splitting control 242
8.16 Quadruples 244
8.17 Differential elimination methods 249
8.18 The Low Power Theorem 250

9 The bas library 253

9.1 Introduction 253
9.1.1 Basic types, constants and naming conventions 253
9.2 Global variables and settings 253
9.3 Bibliography 253
9.4 Leaf contexts 254
9.5 Tasks 257
9.6 Hurwitz coefficients 262

9.7 Pairs 262
9.8 DL Trees 265
9.8.1 Edges 265
9.8.2 Vertices 267
9.8.3 Trees 268

10 The bai library 271
10.1 Introduction 271
10.1.1 Basic types, constants and naming conventions 271
10.2 Variable bindings 271
10.3 Left hand side, mass and jacobian matrices 275
10.4 Recursive polynomials 275
10.5 The RADAU5 DAE integrator 278
10.5.1 Initial values 278
10.5.2 Tolerances 278
10.5.3 Input 279
10.5.4 Result 279
10.5.5 Integration state 280
10.5.6 Dense output 281
10.5.7 Restore point 282
10.5.8 Session 284
10.5.9 Solution 285

Acknowledgements 287

Function Index 288

Type Index 301

1 Introduction

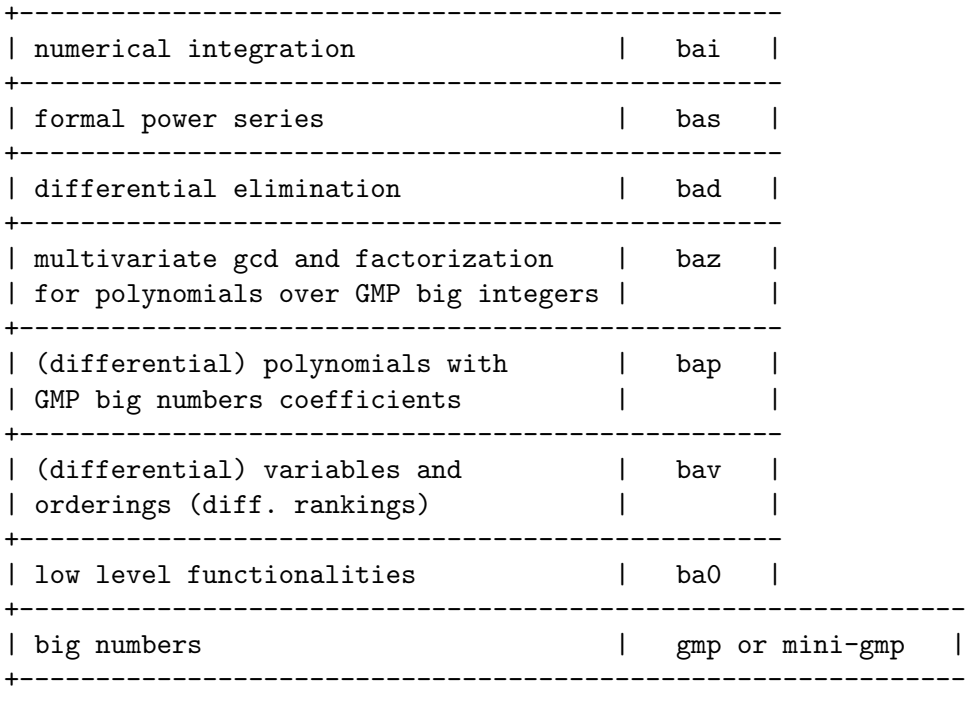
1.1 Introduction (2023)

The BLAD libraries were initially designed as a C library providing differential elimination facilities to C/C++ software. Though this is actually possible, it is much more clear now that they are mostly used through computer algebra software such as Maple and Python/sympy.

For this reason, they are now a part of a bigger project, the DifferentialAlgebra project, hosted on codeberg.org, which involves BLAD but also BMI: the library which provides the interface between BLAD and computer algebra software.

Many examples and introductory texts on Differential Algebra are now available in the gallery directory of the DifferentialAlgebra project.

1.1.1 Structure of the libraries



2 Configuration and Building

The BLAD libraries copying conditions

The BLAD libraries are *free*, which means that everyone is free to use them and free to redistribute them. The libraries are not in the public domain: they are copyrighted and there are restrictions on their distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of these libraries that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the libraries, that you receive source code or else can get it if you want it, that you can change these libraries or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the BLAD libraries, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the BLAD libraries. If they are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the license for the BLAD libraries are found in the Lesser General Public License version 2.1 that accompanies the source code, see `COPYING`.

2.1 Building the development version

The `git` versioning system does not preserve time stamps. For this reason, if you download the development version of `BLAD` through a `git clone` command, you need the `autotools`. Have a look to `tools/autogen` and `tools/autogen_clean`.

```
#!/bin/sh
# autogen - Run the autotools in order to create 'configure'
# FranÂ§ois Boulier. July 2004, 2015, 2025.

if [ ! -f configure.ac ] ; then
   echo "./configure.ac is missing" >&2
   echo "This script must be run in the project top src directory" >&2
   echo "Exiting"
   exit 1
fi

if [ ! -d config ] ; then
    echo mkdir config
    mkdir config
fi

echo "libtoolize --force --copy"
libtoolize --force --copy

if [ $? -ne 0 ] ; then
```

```
# On Mac OS use glibtoolize rather than libtoolize
    echo "libtoolize has failed: trying glibtoolize" >&2
    echo "glibtoolize --force --copy"
    glibtoolize --force --copy
    if [ $? -eq 0 ] ; then
        echo "success" >&2
    else
        echo "glibtoolize has failed also: exiting" >&2
        exit 1
    fi
fi

set -e

echo "aclocal"
aclocal
echo "autoheader"
autoheader
echo "automake --add-missing --copy"
automake --add-missing --copy
echo "autoconf"
autoconf
```

2.2 Building the BLAD libraries

The `BLAD` libraries (bibliothèques lilloises d'algèbre différentielle) rely either on the GNU MP (`GMP`) library or on an embedded version of `mini-gmp`. See `http://gmplib.org`.

They have an `autoconf/automake/libtool` based configuration system. All the usual `autoconf configure` options are available. Run `./configure --help` for a summary.

2.2.1 Recommended build on a Linux machine

Let us assume that the `blad` directory is a subdirectory of `${HOME}/DifferentialAlgebra`. It is recommended to build `BLAD` in a subdirectory `DifferentialAlgebra/blad-build`.

The `tools` subdirectory of `blad` contains a few tools including calls to `configure`. Thus, on a Linux machine, from the `DifferentialAlgebra/blad-build` directory, you may try:

```
../blad/tools/blad_configure_for_install
make
make check
make pdf
make install
```

The last command installs `BLAD` in `DifferentialAlgebra/blad-install`.

2.2.2 Basic build

Enter the following commands from the directory (usually `blad`) which contains the source code:

```
./configure
make
make check (optional)
make dvi (optional)
make install
```

2.2.3 GMP

By default, GMP is used. If it is installed in some non-standard directory, such as gmp-dir, it may be useful to explain to the compiler where to find it. This can be achieved using the --with-gmp facility:

```
configure --with-gmp="gmp-dir"
```

2.2.4 mini-gmp

It is possible to avoid GMP and use the embedded version of mini-gmp instead. This can be achieved using the --enable-gmp=no facility:

```
configure --enable-gmp=no
```

The mini-gmp source code is shipped with GMP. For technical reasons, the source code is adapted and included in the ba0/src/mini-gmp directory. All GMP identifiers are renamed by adding the bam_ prefix to them to avoid name collision. Moreover, the signatures of some of the mini-gmp functions have been redefined by replacing the long type by the BLAD ba0_int_p type, because of issues with MS-Windows. The precise modifications brought to the original mini-gmp source code are given in the sedcmds/to_bam.sed sed command file.

2.2.5 Building example under Darwin

In this example, the GMP library is installed in /usr/local/gmp and we want to install the BLAD libraries in /usr/local/blad. The -m64 needs to be provided to the C and the C++ compilers.

```
./configure CFLAGS="-m64" CXXFLAGS="-m64" LDFLAGS="-m64" \
            --with-gmp="/usr/local/gmp" \
            --prefix="/usr/local/blad"
```

2.2.6 Miscellaneous

2.2.6.1 Tests for debugging purposes

Some computationally intensive debugging tests are implemented in the libraries. They are disabled by default. To enable them, specify the option -DBA0_HEAVY_DEBUG among the CPPFLAGS at configure time.

Another test, cheaper, tests for memory overflows. To enable it, specify the option -DBA0_MEMCHECK among the CPPFLAGS at configure time.

2.2.6.2 Apple Universal OS X

It is possible to generate Apple Universal OS X code on Darwin. In that case, one needs to set BA0_APPLE_UNIVERSAL_OSX among the CPPFLAGS at configure time.

2.2.6.3 32 bits and 64 bits versions

The macro BA0_64BITS is automatically set at configure time. It indicates whether the library is built for a 64-bits system.

To build a 32-bits library on a 64-bits machine, it is sufficient to provide the CFLAGS and LDFLAGS options that produce 32-bits executables.

It is possible (though hardly useful) to force the value of BA0_64BITS when using the BLAD libraries. This can be achieved by providing -DBA0_FORCE_32BITS or -DBA0_FORCE_64BITS among the CPPFLAGS.

2.2.7 What gets installed

The install directory involves four subdirectories:

- include which contains blad.h;
- lib which contains the compiled BLAD library, for static or dynamic linking;
- share which contains info files;
- src which contains all source and header files, ready for further compilation.

This src directory makes it simpler to let external software such as Python/Cython setup tools recompile the whole library with their specific compilation options.

2.2.8 Using the libraries after installation

2.2.8.1 A small C program using BLAD

The following code provides a small C program calling BLAD functions.

```
/* file foo.c */

#include <blad.h>

int main ()
{
    struct ba0_tableof_string T;

    ba0_restart (0, 0);
    ba0_init_table ((ba0_table) &T);
    ba0_sscanf2 ("[this, is, a, table, of, strings]", "%t[%s]", &T);
    ba0_printf ("%t[%s]\n", &T);
    ba0_terminate (ba0_init_level);

    printf ("done\n");
    return 0;
}
```

Let us assume that we are on a Linux machine, using gcc as the compiler and that the BLAD libraries are installed in the directory blad-install. Then the compilation command writes:

```
$ gcc foo.c -Iblad-install/include -Lblad-install/lib \
            -Wl,-rpath -Wl,blad-install/lib -lblad -lgmp
$ a.out
[this, is, a, table, of, strings]
done
```

2.2.8.2 A small C++ program using BLAD

The following code is a C++ translation of the above C program. Observe that the variable declaration of `T` is not the same as in the C program: the `struct` and `ba0_tableof_string` are glued by an underscore: `"_"`.

```
// file foo.cc

#include <iostream>
#include <blad.h>

int main ()
{
    ba0_restart (0, 0);

    struct ba0_tableof_string T;

    ba0_init_table ((ba0_table)&T);
    ba0_sscanf2 ("[this, is, a, table, of, strings]", "%t[%s]", &T);
    ba0_printf ("%t[%s]\n", &T);
    ba0_terminate (ba0_init_level);

    std::cout << "done\n";
    return 0;
}
```

Let us assume that we are on a Linux machine, using `g++` and that the `BLAD` libraries are installed in the directory `blad-install`. Then the compilation command writes:

```
$ g++ foo.cc -Iblad-install/include -Lblad-install/lib \
             -Wl,-rpath -Wl,blad-install/lib -lblad -lgmp
$ a.out
[this, is, a, table, of, strings]
done
```

2.2.8.3 Caveats

Let's assume that the `BLAD` libraries are installed in a directory called `blad-dir`. Let's assume that you want to produce an executable from a C source file without using `libtool`. Let's assume you are using `gcc` as the compiler.

- if the C preprocessor does not find the header files, you may try to add the option `-Iblad-dir/include` to the `gcc` command line.
- if the linker does not find the libraries, you may try to add the option `-Lblad-dir/lib` and possibly `-Wl,-rpath -Wl,blad-dir/lib` to the `gcc` command line.
- if the linker does not find some symbols, take care to the order of the libraries on the `gcc` command line. The right order is: `-lblad -lgmp -lm`.

2.2.9 Reporting bugs

If you think you have found a bug in one of the libraries, please investigate and report it.

Please include in the bug report:

- The version of `BLAD`.
- A test program that makes it possible to reproduce the bug. Include instructions on how to run the program.
- A description of what is wrong. If the results are incorrect, in what way.
- Please do not send core dumps or executables.
- The configuration options (see the top of config.log).
- The version of `GMP` that you use.
- The output from `uname -a`.

Send your report to Francois.Boulier [at] univ-lille.fr

There are probably many bugs in this first version of the libraries. I apologize for them. I will try to solve them as fast as possible but even this may take time!

2.3 Building BLAD under MS-Windows

The `BLAD` libraries can be compiled using Microsoft C Compiler. Indeed, the Windows Python wheels for the `DifferentialAlgebra` package are obtained by compiling the `BLAD` C code, via Python facilities.

3 The developer corner

3.1 The tools directory

The `tools` directory contains various shell scripts which are useful for developers only. For portability reasons, they rely as much as possible on a restricted set of `sed` commands.

Many of these `sed` commands run on the source code to extract information. They rely on the fact that all `BLAD` source and header files are formatted using the tool `blad_indent`.

The tool `blad_data_types` extracts all the type definitions from the header files. The tool `blad_extern_functions` extracts all the extern/exported functions from the header files too.

Each library has at least one `tests` subdirectory. Each `tests` directory contains a `coverage` subdirectory which contains one text file per test (with the same name). Each text file contains the names of all `BLAD` extern functions run by the test. These text files are automatically generated by a tool called `blad_test_coverage`. In the future, this shell script might be run from the `Makefile`, but for the moment it is not. It is not only heavy but its portability is not clear. In particular, it might call another tool called `blad_configure_for_test_coverage` and it relies on the `clang` compiler and on `llvm`. For this reason, its output is included in the git repository.

A tool called `blad_function_coverage` runs over these text files and provides, for each extern function, the tests which run it.

A tool called `blad_doc_function_extract` builds a `.texi` file per extern function. These files are stored in the subdirectory `doc/functions` of each `BLAD` library. The tool extracts the prototype of the function directly from the `BLAD` source files. It also extracts a comment (if any) starting by the `texinfo:` tag followed by the function identifier from the source file. Moreover, it adds to the `.texi` file some further information: the name of the source file which contains the function and some statistics on the tests which run it. The output of this function is included in the git repository.

Lastly a tool called `blad_doc_type_extract` builds a `.texi` file per data type. These files are stored in the subdirectory `doc/types` of each `BLAD` library. The output of this function is included in the git repository.

3.2 Implementing a new library

Let us take the example of the most recent one: the `bas` library, which is inserted below the `bai` library and above the `bad` library.

Create `bas/src`, `bas/tests` and `bas/doc`. Add the four following entries to the `AC_CONFIG_FILES` variable of `configure.ac`.

```
bas/Makefile
bas/src/Makefile
bas/doc/Makefile
bas/tests/Makefile
```

Add `bas` at the right place in the `sublibraries` variable of `Makefile.am`. Copy `bai/Makefile.am` in `bas` and ensure that the `SUBDIRS` variable only involves the subdirectories actually present in `bas`.

3.2.1 The src directory

The two files bas_common.h and bas_common.c and a Makefile.am need be created in bas/src. Create them by copying the corresponding ones from the bai library.

In each of these files, replace any occurrence of BAI by BAS and bai by bas. Simplify the module bas_common and ensure that the bas_restart, bas_terminate and bas_reset_all_settings are present and actually call the corresponding functions of the bad library. In the variables header_files and source_files of Makefile.am, ensure that only bas_common.h and bas_common.c occur.

3.2.2 The doc directory

Create bas.texi, bas-core.texi and Makefile.am in bas/doc. Create them by copying the corresponding ones from the bai library.

Replace any occurrence of bai by bas in all these files. Simplify bai-core.texi but keep inclusion directives for bas_restart.texi, bas_terminate.texi and bas_reset_all_settings.texi.

Create the two directories bas/doc/functions and bas/doc/types.

3.2.3 The tests directory

Create the following minimal just_link.c test.

```
#include "bas.h"

int
main (
    void)
{
  bas_reset_all_settings ();
  bas_restart (0, 0);
  bas_terminate (ba0_init_level);
  return 0;
}
```

Create Makefile.am from the one of bai/tests/Makefile.am. Replace any occurrence of bai by bas in this file and simplify it so that it only lists just_link.c. Beware to update the AM_CPPFLAGS, AM_LDFLAGS and LDADD variables.

3.2.4 Modifications to the other libraries

Add -I$(top_builddir)/bas/src to the AM_CPPFLAGS variable of bai/src/Makefile.am. Include bas.h instead of bad.h in bai/src/bai_common.h. Replace the calls to bad_restart, bad_terminate and bad_reset_all_settings by calls to bas_restart, bas_terminate and bas_reset_all_settings in the file bai/src/bai_common.c.

In the Makefile.am of all bai/tests directories, add -I$(top_builddir)/bas/src in the AM_CPPFLAGS variable, add -L$(top_builddir)/bas/src in the AM_LDFLAGS variable and add $(top_builddir)/bas/src/libbas.la to the LDADD variable.

In the big/Makefile.am add bas.h and bas_common.h to the header_files variable and bas_common.c to the source_files variable. Add $(top_builddir)/bas/src/bas.h to the dependencies of blad.h and update the actions to perform.

In `doc/Makefile.am` update the variables `blad_TEXINFOS` and `AM_MAKEINFOFLAGS`. In this `doc/blad-core.texi` file, add an inclusion directive to `bas-core.texi`.

3.2.5 Building the library

From the `blad` directory run the following commands which generate `.texi` files which are mandatory.

```
$ ./tools/doc_type_extract bas -top_srcdir=.
$ ./tools/doc_function_extract bas -top_srcdir=.
```

From the `blad-build` directory, which is supposed to be at the same level as the `blad` one, run a `configure` command such as

```
$ ../blad/tools/blad_configure_for_install
```

Then

```
$ cd bas
$ make
$ make check
$ make pdf
```

Then

```
$ cd ../bai
$ make
$ make check
$ make pdf
```

If all these steps work, build the whole documentation. From the `blad` library run:

```
$ doc_generate -top_srcdir=.
```

4 The ba0 library

4.1 Introduction

The ba0 library implements low level functionalities for all the libraries of the blad project: memory management, exception handling, input/output, garbage collection. It also implements some general purpose data structures: lists, tables and matrices. For big numbers, it relies either on the GMP library or on an embedded version of mini-gmp.

This manual describes key features of the library.

4.1.1 Basic types, constants and naming conventions

The name of every exported object defined in the library starts by the prefix ba0_. Functions starting by ba0_init initialize raw data structures (constructors). Functions starting by ba0_new dynamically allocate data structures and apply the corresponding ba0_init over them. Functions starting by ba0_reset reset data structures which are assumed to be already initialized. Functions starting by ba0_set perform assignments and systematically copy the sources to the destinations. Here are a few types and constants. Some other ones can be found in the file ba0_common.h.

- ba0_int_p is an integer type
- ba0_int_hp is an integer type which has half the size of a ba0_int_p
- ba0_mint_hp is a ba0_int_hp used for modular arithmetic (actually an unsigned ba0_int_hp).
- BA0_MAX_PRIME_MINT_HP is the largest prime integer which fits in a ba0_mint_hp.

4.1.2 Checking the version number of the library

`char * ba0_get_version (void)` [Function]

Return the version of the library as a string. This string is made of a major number + dot + a minor number possibly + dot + a micro number. This function belongs to ba0_common.c. It is not run by any test.

4.1.3 Global variables and settings

Setting variables are global configuration variables which permit to tune the behaviour of the library functions. This tuning is supposed to be performed before ba0_restart is called (more generally, outside any sequence of calls the library). They are bundled in the ba0_initialized_global global variable which has the following type.

Description of type ba0_initialized_global. This data type is used by a single global variable. It is decomposed into fields. The values it contains permit to tune the library behaviour. This type declaration belongs to ba0_global.h.

```
struct ba0_initialized_global
{
  struct
  {
/*
 * The only settings variable which is modified by ba0_restart/ba0_terminate.
 * Set by ba0_terminate. Read by all ba__restart functions.
```

```
 */
    enum ba0_restart_level restart_level;
/*
 * check_interrupt = the pointer to the external check interrupt function,
 *                   called by ba0_process_check_interrupt.
 * delay_interrupt = the delay (in sec.) between two calls to *check_interrupt.
 * no_oot          = if true, ERRALR is disabled.
 */
    void (
        *check_interrupt) (
        void);
    time_t delay_interrupt;
    bool no_oot;
  } common;
  struct
  {
/*
 * If true, the BLAD memory limit does not raise BA0_ERROOM.
 * Temporarily modified by all blad_restart functions.
 */
    bool no_oom;
  } malloc;
  struct
  {
    ba0_int_p sizeof_main_cell;
    ba0_int_p sizeof_quiet_cell;
    ba0_int_p sizeof_analex_cell;
    ba0_int_p nb_cells_per_stack;
    ba0_int_p sizeof_stack_of_stack;
/*
 * Pointers to the functions that are called by BLAD for allocating/freeing
 * memory (mostly cells in stacks).
 */
    void *(
        *system_malloc) (
        size_t);
    void (
        *system_free) (
        void *);
  } stack;
  struct
  {
    ba0_set_memory_functions_function *set_memory_functions;
/*
 * If Integer_PFE is not (char *)0, every mpz_t z is printed Integer_PFE(z).
 * This mechanism prevents floating point evaluation of rational numbers
 *  in Python. Default value is (char *)0
 */
    char *Integer_PFE;
  } gmp;
  struct
  {
/*
 * nb_tokens = the number of tokens in the analex FIFO
 * quotes    = a string containing the characters that can be used
 *             for quoting tokens
 */
    ba0_int_p nb_tokens;
```

```
    char *quotes;
  } analex;
  struct
  {
/*
 * The operator used for separating variable from values. Default "="
 */
    char *equal_sign;
  } value;
  struct
  {
/*
 * oper     = the range operator
 * infinity = the symbol for infinity
 * rhs_included = determines whether the right-hand side of range
 *                indices should be included or not in the ranges
 * quote_PFE = if set to true, the range indexed groups which are not plain
 *                strings are quoted when printed; this mechanism prevents
 *                from evaluation failures in Python; default (char *)0
 */
    char *oper;
    char *infinity;
    bool rhs_included;
    bool quote_PFE;
  } range_indexed_group;
};
```

Some global variables are read and modified during sequences of calls to the library. They are bundled in the `ba0_global` variable which has the following type.

Description of type `ba0_global`. This data type is used for a single global variable. Each field aims at customizing the behaviour of the `ba0` library. This type declaration belongs to `ba0_global.h`.

```
struct ba0_global
{
  struct
  {
/*
 * Deprecated control for printing in LaTeX.
 * Used in ba0, bav and bap.
 */
    bool LaTeX;
/*
 * Time out + Memory out + Interrupt checking
 *
 * time_limit           = the absolute value of the input time limit (restart)
 * memory_limit         = the input memory limit (restart)
 * switch_on_interrupt  = *check_interrupt should be called
 * within_interrupt     = bool to avoid self interruption
 * delay_interrupt      = the length of a time interval between two calls
 * before_timeout       = the overall remaining time before timeout
 * previous_time        = the value of time() when interrupt was last called
 */
    ba0_int_p time_limit;       /* local to ba0_common */
    ba0_int_p memory_limit;     /* local to ba0_common and ba0_stack */
    bool switch_on_interrupt;   /* local to ba0_common */
    bool within_interrupt;      /* local to ba0_common */
    time_t before_timeout;      /* local to ba0_common */
```

```
    time_t previous_time;       /* local to ba0_common */
  } common;
  struct
  {
/*
 * The error/exception message.
 * Only set when raising an exception. Read everywhere.
 */
    char *raised;
    char mesg_cerr[BA0_BUFSIZE];
/*
 * The stack of exception catching points.
 * Only set by the exception setting/raising MACROS.
 */
    struct
    {
      struct ba0_exception tab[BA0_SIZE_EXCEPTION_STACK];
      ba0_int_p size;
    } stack;
/*
 * The stack of extra variables to be saved/restored when setting
 *      an exception point/raising an exception.
 * The field pointer points to the extra variable to be restored.
 * If the field restore is nonzero then it points to a function which is
 *      called in order to restore the extra variable with the value passed
 *      as a parameter.
 * The saved value is stored in a local variable at the catching point.
 */
    struct
    {
      struct
      {
        ba0_int_p *pointer;
        void (
            *restore) (
            ba0_int_p);
      } tab[BA0_SIZE_EXCEPTION_EXTRA_STACK];
      ba0_int_p size;
    } extra_stack;
/*
 * bool to avoid self interruption
 */
    bool within_push_exception;
/*
 * A log fifo for debugging purposes
 * The entries of tab provide the sequence of file/line/exception raised
 *      since the last exception catching point was set.
 * The field qp contains the first free entry in tab
 */
    struct
    {
      struct
      {
        char *file;
        int line;
        char *raised;
      } tab[BA0_SIZE_EXCEPTION_LOG];
      ba0_int_p qp;
```

```
    } log;
  } exception;
  struct
  {
/*
 * The garbage collector
 * tab                = an array for old->new addresses of areas
 * user_provided_mark = the mark M provided by garbage (format, M, ...)
 * ba0_current        = a running mark on struct ba0_gc_info(s)
 * old_free           = the value of the free pointer when garbage is called.
 *
 * All local to ba0_garbage.
 * The values kept between two calls to ba0_garbage are meaningless.
 */
    struct ba0_gc_info **tab;
    struct ba0_mark user_provided_mark;
    struct ba0_mark current;
    struct ba0_mark old_free;
  } garbage;
  struct
  {
/*
 * The predefined stacks. Used everywhere.
 */
    struct ba0_stack main;
    struct ba0_stack second;
    struct ba0_stack analex;
    struct ba0_stack quiet;
    struct ba0_stack format;
/*
 * The current stack is the one on the top of stack_of_stacks.
 */
    struct ba0_tableof_stack stack_of_stacks;
/*
 * For stats and debugging purposes.
 * Local to ba0_common, ba0_stack and ba0_analex.
 */
    ba0_int_p alloc_counter;
    ba0_int_p malloc_counter;
    ba0_int_p malloc_nbcalls;
  } stack;
  struct
  {
/*
 * atomic_format_component = The formats defined by the library, such as %s, %d ...
 *                   Non-leaf formats are %t (tables) or %l (lists), .
 * htable         = The H-table of all the encountered formats.
 *                   Its size is a prime number.
 * nbelem_htable  = The number of its elements.
 *
 * All are local to ba0_format.
 */
    struct ba0_tableof_pair atomic_format_component;
    struct ba0_tableof_pair htable;
    ba0_int_p nbelem_htable;
/*
 * Management of the variables (scanf, printf) for the %value format.
 * Eventually, these pointers will point to bav_scanf_variable and
```

```
 * bav_printf_variable ... whenever these functions are defined (in bav).
 * Meanwhile, calling them raises BA0_ERRNYP
 */
    ba0_scanf_function *scanf_value_var;
    ba0_printf_function *printf_value_var;
  } format;
  struct
  {
/*
 * output, input      = descriptions of the device in use.
 * output_line_length = to insert carriage returns.
 *                      Reset by ba0_restart. Can be modified at runtime.
 *
 * All variables (except output_line_length) are local to ba0_basic_io.
 */
    struct ba0_output_device output_stack[BA0_BASIC_IO_SIZE_STACK];
    ba0_int_p output_sp;
    struct ba0_output_device output;
    ba0_int_p output_line_length;

    struct ba0_input_device input_stack[BA0_BASIC_IO_SIZE_STACK];
    ba0_int_p input_sp;
    struct ba0_input_device input;
  } basic_io;
  struct
  {
/*
 * analex           = the FIFO of tokens
 * analex_save      = permits to record/restore analex (up to some point)
 * analex_save_full = indicates if analex_save is used
 * context          = an elaborated error message for parser errors
 *
 * subs_dict        = a pointer to a dictionary for performing token substitutions
 * subs_keys        = a pointer to the table which contains the dictionary keys
 * subs_vals        = a pointer to the table which contains the dictionary values
 */
    bool analex_save_full;
    struct ba0_analex_token_fifo analex;
    struct ba0_analex_token_fifo analex_save;

    char context[BA0_CONTEXT_LMAX];

    struct ba0_dictionary_string *subs_dict;
    struct ba0_tableof_string *subs_keys;
    struct ba0_tableof_string *subs_vals;
  } analex;
  struct
  {
/*
 * These variables receive the values of the GMP memory functions before
 * they get modified by BLAD.
 *
 * Set by ba0_restart. Read by ba0_terminate.
 * Set by ba0_process_check_interrupt before calling *check_interrupt
 * and restored afterwards.
 */
    bool alloc_function_called;
    void *(
```

```
        *alloc_function) (
        size_t);
    void *(
        *realloc_function) (
        void *,
        size_t,
        size_t);
    void (
        *free_function) (
        void *,
        size_t);
  } gmp;
  struct
  {
/*
 * For computing in Z / module Z. Read everywhere.
 */
    bool module_is_prime;
    unsigned ba0_int_hp module;
  } mint_hp;
  struct
  {
/*
 * For computing in Z / module Z with a GMP module. Read everywhere.
 */
    bool module_is_prime;
    ba0_mpz_t module;
    ba0_mpz_t half_module;
    ba0_mpz_t accum;
  } mpzm;
};
```

`void ba0_reset_all_settings (void)` [Function]

Reset all settings to their default values. Must be called outside sequences of calls to the library. This function belongs to `ba0_common.c`. It is run by 65 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_settings_no_oot (bool b)` [Function]

Set to b a settings variable which permits to forbid the exception `BA0_ERRALR` (out of time). This settings function can be called within sequences of calls to the library, in order to prevent critical code to be interrupted. This function belongs to `ba0_common.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_no_oot (bool *b)` [Function]

Assign to $*b$, the value of the settings variable described above. This function belongs to `ba0_common.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_settings_no_oom (bool b)` [Function]

Set to b a settings variable which permits to forbid the exception `BA0_ERROOM` (out of memory) when possible. This settings function can be called within sequences of calls to the library, in order to prevent critical code to be interrupted. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_no_oom (bool *b)` [Function]
Assign to *b, the value of the settings variable described above. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

Description of type `ba0_PFE_settings`. This data structure is used to store copies of the settings variables contained in the fields of `ba0_initialized_global` which contain `PFE` variables. This type declaration belongs to `ba0_global.h`.

```
struct ba0_PFE_settings
{
  ba0_set_memory_functions_function *set_memory_functions;
  char *Integer_PFE;
  char *oper;
  char *infinity;
  bool rhs_included;
  bool quote_PFE;
};
```

`void ba0_cancel_PFE_settings (struct ba0_PFE_settings *P)` [Function]
Fill the fields of P with the corresponding settings variables. Then cancel the `PFE` settings. This function belongs to `ba0_common.c`. It is run by 2 tests, including 0 in the `ba0` library and 2 in the `DifferentialAlgebra` package.

`void ba0_restore_PFE_settings (struct ba0_PFE_settings *P)` [Function]
Restore the `PFE` settings with the content of P. This function belongs to `ba0_common.c`. It is run by 2 tests, including 0 in the `ba0` library and 2 in the `DifferentialAlgebra` package.

4.1.4 Sequences of calls to the library

Every *sequence of calls* to the `ba0` library functions should start with a call to `ba0_restart` and finish with a call to `ba0_terminate`. The only functions that can be executed outside any sequence of calls are the *settings functions*.

`void ba0_restart (ba0_int_p time_limit, ba0_int_p memory_limit)` [Function]
The parameter *memory_limit* provides the maximal number of megabytes authorized for the coming sequence of calls. The value may be augmented to the minimal reasonable value if it is too small. A value zero means no limit. If the bound is reached, the exception `BA0_ERROOM` is raised.

The parameter *time_limit* is a number of seconds. If it is nonnegative then, it provides the maximal number of seconds authorized for the coming sequence of calls. A value zero means no limit. If the bound is reached, the exception `BA0_ERRALR` is raised.

If *time_limit* is negative, then its opposite provides the maximal number of seconds authorized for the coming sequence of calls. Moreover, an interrupt function is called at fixed time intervals (see `ba0_set_settings_interrupt`).

This function initializes and resets all non settings global variables. In particular, it redirects the `GMP` memory management functions so that they allocate memory in the BLAD stacks. This function belongs to `ba0_common.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_terminate (enum ba0_restart_level level)` [Function]

A call to this function terminates a sequence of calls to the `ba0` library. The parameter *level* permits to control the level of cleaning of the resources used by the library (an usual value is `ba0_init_level`). The value of the parameter is stored in the global variable `ba0_initialized_global.common.restart_level` which will be used by `ba0_restart` at the next call.

- `ba0_done_level`

  Does not perform anything.

- `ba0_reset_level`

  Empties the stacks `ba0_global.stack.main`, `ba0_global.stack.second`, `ba0_global.stack.quiet` and `ba0_global.stack.analex` and resets the `GMP` memory management functions to the values they had before the call to `ba0_restart`.

- `ba0_init_level`

  Frees all the resources used by the sequence of calls.

If the total number of calls to `ba0_free` is different from the total number of calls to `ba0_malloc` then the exception `BA0_ERRMFR` is raised. This function belongs to `ba0_common.c`. It is run by 547 tests, including 53 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_settings_common (enum ba0_restart_level level)` [Function]

Assign *level* to `ba0_initialized_global.common.restart_level`. The parameter may be zero. This function belongs to `ba0_common.c`. It is run by 65 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_common (enum ba0_restart_level *level)` [Function]

Assign `ba0_initialized_global.common.restart_level` to **level*. The parameter may be zero. This function belongs to `ba0_common.c`. It is not run by any test.

`void ba0_set_settings_interrupt (void (*check) (void), time_t delay)` [Function]

Set the *interrupt* function to *check*. For any following sequence of calls starting by a call to `ba0_restart` with a negative *time_limit* value, the *check* function will be called at time interval of length *delay*.

The whole interrupt mechanism can be cancelled by providing a zero value for *check*.

This function must be modified outside sequences of calls to the library. This function belongs to `ba0_common.c`. It is run by 65 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_interrupt (void (**check) (void), time_t *delay)` [Function]

Assign to *check* and *delay* the current values of the corresponding variables. Parameters may be zero. This function belongs to `ba0_common.c`. It is not run by any test.

4.2 Memory management

4.2.1 Overview

The library manages its own memory, which is subdivided in *stacks* with the aim of providing a garbage collection mechanism as well as an exception handling mechanism. The `malloc` and `free` standard functions should thus only be used in very rare and specific cases. The GMP memory allocation functions are normally modified in order to allocate memory within BLAD stacks.

4.2.2 System memory allocation

The following functions provide an interface to the system `malloc` and `free` functions.

`void * ba0_malloc (ba0_int_p n)` [Function]
Allocate at least n bytes by calling the system `malloc`. The allocated memory is eventually freed by a call to `ba0_free` issued by `ba0_terminate`. This function (and the next one) are the only one allowed to call the system `malloc`. They are also the only ones which check the memory limit. They may raise exception `BA0_ERROOM`. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void * ba0_persistent_malloc (ba0_int_p n)` [Function]
Allocate at least n bytes by calling the system `malloc`. The allocated memory is not freed by any call to `ba0_free` issued by `ba0_terminate`. Exception `BA0_ERROOM` may be raised. This function is designed to allocate areas which should be kept after calling `ba0_terminate`. See `ba0_new_printf`. This function belongs to `ba0_stack.c`. It is run by 224 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_free (void *p)` [Function]
Free the area pointed to by p, which is assumed to be allocated through `ba0_malloc`. This function belongs to `ba0_stack.c`. It is run by 547 tests, including 53 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_settings_memory_functions (void *(*m) (size_t), void (*f) (void *))` [Function]
Make the three above function call *malloc* and *free* instead of the system `malloc` and `free` functions. This function belongs to `ba0_stack.c`. It is run by 65 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_memory_functions (void *(**m) (size_t), void (**f) (void *))` [Function]
Assign to *malloc* and *free* the addresses of the functions called by `ba0_malloc` and `ba0_free` for allocating and freeing memory. This function belongs to `ba0_stack.c`. It is not run by any test.

4.2.3 GMP memory allocation

The following functions permit to tune GMP memory allocation functions.

`void ba0_set_settings_gmp` [Function]
`(ba0_set_memory_functions_function *set, char *Integer_PFE)`

Set to *set* the settings variable corresponding to the `mpz_set_memory_function` function. Set to *Integer_PFE* the corresponding settings variable. Every zero argument resets the settings variable to its default value. This function belongs to `ba0_gmp.c`. It is run by 502 tests, including 15 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_gmp` [Function]
`(ba0_set_memory_functions_function **set, char **Integer_PFE)`

Assign to its parameters the values of the corresponding fields in the `gmp` substructure of `ba0_initialized_global`. Zero parameters are allowed. This function belongs to `ba0_gmp.c`. It is run by 502 tests, including 15 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.2.4 Stacks

Description of type `ba0_mark`. The data structure `ba0_mark` implements marks. From the user point of view, a *mark* is a pointer in a stack. Marks are used for garbage collection. This type declaration belongs to `ba0_stack.h`.

```
struct ba0_mark
{
// the stack the address belongs to
  struct ba0_stack *stack;      // the stack the address belongs to
// the index of the stack cell the address belongs to
  ba0_int_p index_in_cells;
  void *address;                // the address
// the number of free bytes in the cell, after the mark
  unsigned ba0_int_p memory_left;
};
```

Description of type `ba0_stack`. The data structure `ba0_stack` mplements stacks. From the user point of view, a stack is a huge piece of memory in which memory allocation is possible. A stack is implemented as an array of cells. Cells are blocks of memory allocated in the process heap via `ba0_alloc`. This type declaration belongs to `ba0_stack.h`.

```
struct ba0_stack
{
// the identifier (nickname) of the stack
  char *ident;
// the array of cells and the corresponding array of cell sizes
  struct ba0_table cells;
  struct ba0_tableof_int_p sizes;
// the default size of cells (may increase during computations)
  unsigned ba0_int_p default_size;
// default value is true (if false, cell sizes do not increase)
  bool resizable;
// the border between the used and the free parts of the stack
  struct ba0_mark free;
// the max address reached by the free pointer
  struct ba0_mark max_alloc;
// counts the number of allocations in the process heap
  ba0_int_p nb_calls_to_alloc;
// for debugging purposes
  ba0_int_p *bound;
```

```
};
```

Five stacks are predefined in the library: `ba0_main_stack`, `ba0_second_stack` (two general purpose stacks), `ba0_analex_stack` (private stack for the lexical analyzer), `ba0_quiet_stack` (a stack over which no garbage collection is performed) and `ba0_format_stack` (private stack for *formats*). The stack `ba0_analex_stack` is managed in a special way by the lexical analyzer.

A *stack of stacks* is implemented also. Its elements are pointers to stacks. Its top element is the *current stack*: the one in which BLAD memory allocation functions implicitly apply.

4.2.4.1 Stack settings

`void ba0_set_settings_stack (ba0_int_p` *`main`*`, ba0_int_p` *`quiet`*`, ba0_int_p` *`analex`*`, ba0_int_p` *`nbcells`*`, ba0_int_p` *`sizess`*`)` [Function]

Set the initial default cell sizes of `ba0_global.stack.main` and `ba0_global.stack.second` to *main*, those of `ba0_global.stack.quiet` and `ba0_global.stack.format` to *quiet* and those of `ba0_global.stack.analex` to *analex*. Set the initial number of cells per stack to *nbcells*. Set the initial size of the stack of stacks to *sizess*. If one of these arguments is zero, a default value is used. This function belongs to `ba0_stack.c`. It is run by 65 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_stack (ba0_int_p *`*`main`*`, ba0_int_p *`*`quiet`*`, ba0_int_p *`*`analex`*`, ba0_int_p *`*`nbcells`*`, ba0_int_p *`*`sizess`*`)` [Function]

Assign to **main*, **quiet* and **analex* the current values used for allocating cell sizes. Assign to *nbcells* the number of cells per stack and to *sizess* the size of the stack of stacks. The pointers are allowed to be zero. This function belongs to `ba0_stack.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.2.4.2 Memory alignment

`unsigned ba0_int_p ba0_ceil_align (unsigned ba0_int_p` *`n`*`)` [Function]

Round *size* in order to get a multiple of the `BA0_ALIGN` constant. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_allocated_size (unsigned ba0_int_p` *`n`*`)` [Function]

Return the number of bytes that `ba0_alloc` would allocate for *size* bytes. This function belongs to `ba0_stack.c`. It is run by 154 tests, including 28 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.2.4.3 Stack primitives

`void ba0_init_stack (struct ba0_stack *`*`H`*`)` [Function]

Initialize the stack *H*. The field `resizable` is set to true. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_stack (struct ba0_stack *H)` [Function]
Empty the stack H but does not free the allocated cells. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_cell_stack (struct ba0_stack *H)` [Function]
Empty the current cell of the stack H. This function belongs to `ba0_stack.c`. It is run by 531 tests, including 41 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_clear_cells_stack (struct ba0_stack *H)` [Function]
Empty the stack H and frees the allocated cells. This function belongs to `ba0_stack.c`. It is run by 547 tests, including 53 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_clear_stack (struct ba0_stack *H)` [Function]
Empty the stack H, free the allocated cells and the arrays `cells.tab` and `sizes.tab` of stack H. This function belongs to `ba0_stack.c`. It is run by 547 tests, including 53 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_init_one_cell_stack (struct ba0_stack *H, char *ident, void *cell, ba0_int_p cell_size)` [Function]
Initialize the stack H with the cell *cell*, which is assumed to have size *cell_size*. This stack can afterwards be used as any other stack. The stack has its field `resizable` set to false, so that, if an extra cell needs to be allocated, then the exception `BA0_ERROOM` is raised. This function belongs to `ba0_stack.c`. It is run by 75 tests, including 1 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_clear_one_cell_stack (struct ba0_stack *H)` [Function]
Free the arrays `cells.tab` and `sizes.tab` the stack H but does not free the unique cell of H. This function belongs to `ba0_stack.c`. It is run by 75 tests, including 1 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`struct ba0_stack * ba0_current_stack (void)` [Function]
Return a pointer to the current stack. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void * ba0_alloc (unsigned ba0_int_p n)` [Function]
Allocate n bytes in the current stack. The number n must be nonzero. A mechanism is implemented which ensures that memory is always aligned. Moreover, the function allocates a bit more memory than n bytes and stores a magic number at the tail of the n bytes. This magic number is used to detect at runtime pointers running out of allocated data: the next call to the function will check that the stored magic numbered is not erased. If it is, an exception is raised. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_memory_left_in_cell (void)` [Function]
Return the amount of memory that could be allocated in the current stack without moving the free pointer to the next cell. This function belongs to `ba0_stack.c`. It is run by 532 tests, including 42 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_t1_alloc (unsigned ba0_int_p se, unsigned ba0_int_p n, void **te, unsigned ba0_int_p *m)` [Function]
One wishes to allocate an array *te* of *n* elements. Each element has size *se*. It may happen that *n* is too large to stand in a cell. This function decreases *n* if needed. The number of elements actually allocated is stored in *m*. This function belongs to `ba0_stack.c`. It is run by 280 tests, including 0 in the `ba0` library and 41 in the `DifferentialAlgebra` package.

`void ba0_t2_alloc (unsigned ba0_int_p sc, unsigned ba0_int_p st, unsigned ba0_int_p n, void **tc, void **tt, unsigned ba0_int_p *m)` [Function]
Variant of the above function. Two arrays of the same number of elements *tc* and *tt* must be allocated. The desired number of elements is *n* which may be decreased if any of the arrays does not fit in a cell. The actual number of allocated elements is stored in *m*. The elements of the arrays have respective sizes *sc* and *st*. This function belongs to `ba0_stack.c`. It is run by 384 tests, including 0 in the `ba0` library and 64 in the `DifferentialAlgebra` package.

`void ba0_rotate_cells (ba0_int_p i)` [Function]
Perform a rotation between the cells of the current stack located at indices greater than or equal to *i*. The cell number *i* gets located at the end of the cells table. The index *i* must be strictly less than the index of the free pointer. This function should be used with a lot of care. It is called by the lexical analyzer. This function belongs to `ba0_stack.c`. It is run by 3 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

Stack debugging primitives

`bool ba0_in_stack (void *p, struct ba0_stack *H)` [Function]
Return `true` if *p* is pointing somewhere in *H* and `false` otherwise. This function belongs to `ba0_stack.c`. It is run by 110 tests, including 1 in the `ba0` library and 8 in the `DifferentialAlgebra` package.

`struct ba0_stack * ba0_which_stack (void *p)` [Function]
Return the address of the stack where *p* is pointing. Return zero if not found. This function only looks in `ba0_global.stack.main`, `ba0_global.stack.second` and `ba0_global.stack.quiet`. This function belongs to `ba0_stack.c`. It is run by 35 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_max_alloc_stack (struct ba0_stack *H)` [Function]
Return the maximal number of bytes which has been used in *H*. This function belongs to `ba0_stack.c`. It is not run by any test.

4.2.4.4 Stack of stacks primitives

`void ba0_init_stack_of_stacks (void)` [Function]
Initialize the stack of stacks array. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_stack_of_stacks (void)` [Function]
Reset `ba0_global.stack.stack_of_stacks` and push `ba0_global.stack.main` which then becomes the current stack. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_clear_stack_of_stacks (void)` [Function]
Free the area allocated to `ba0_global.stack.stack_of_stacks`. This function belongs to `ba0_stack.c`. It is run by 547 tests, including 53 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_push_stack (struct ba0_stack *`*H*`)` [Function]
Push the stack H on the top of `ba0_global.stack.stack_of_stacks`. Then H becomes the new current stack. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_pull_stack (void)` [Function]
Undo the previous call to `ba0_push_stack`. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_push_another_stack (void)` [Function]
Push on the top of the stack of stacks a stack which is different from the current one. The chosen stack is either `ba0_global.stack.main` of `ba0_global.stack.second`. This function belongs to `ba0_stack.c`. It is run by 532 tests, including 42 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.2.4.5 Mark primitives

`ba0_int_p ba0_cell_index_mark (void *`*p*`, struct ba0_mark *`*M*`)` [Function]
The pointer p is assumed to point in the stack referred to by M. The function returns the index, in the `cells` array, of the cell p points to. The function searches between the beginning of the stack and M. Returns -1 if not found. This function belongs to `ba0_stack.c`. It is run by 154 tests, including 25 in the `ba0` library and 8 in the `DifferentialAlgebra` package.

`void * ba0_alloc_mark (struct ba0_mark *`*M*`, unsigned ba0_int_p `*n*`)` [Function]
Simulate an allocation of n bytes by `ba0_alloc` assuming that the free pointer is M. Move M to the new value of the free pointer. This function belongs to `ba0_stack.c`. It is run by 77 tests, including 25 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_record (struct ba0_mark *M)` [Function]

Record in M the value of the free pointer of the current stack. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_restore (struct ba0_mark *M)` [Function]

Restore the value of the free pointer which was stored in M by a previous call to `ba0_record`. This function belongs to `ba0_stack.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_range_mark (struct ba0_mark *A, struct ba0_mark *B)` [Function]

Return the amount of memory enclosed between the two marks. The mark A is assumed pointing to an area before the mark B. This function belongs to `ba0_stack.c`. It is run by 48 tests, including 3 in the `ba0` library and 36 in the `DifferentialAlgebra` package.

4.2.5 Garbage collection

Two garbage collection schemes are provided. Note that the second one makes uses of *formats*, which will be introduced in a following section.

4.2.5.1 The basic scheme

The next example is very basic though technical. A mark is set at the beginning of a computation. All the memory used after the mark is set is freed. Recall that a `ba0_mpz_t` involves a dynamically allocated field named `_mp_d`.

```
#include "ba0.h"

int
main (
    void)
{
  struct ba0_mark M;
  ba0_mpz_t p, q;
  bool b;

  ba0_restart (0, 0);
  ba0_mpz_init_set_si (p, 421); // p = 421
// Mark M is set after p->_mp_d
  ba0_record (&M);
  ba0_mpz_init (q);
  ba0_mpz_add (q, p, p);        // q = p + p
// q->_mp_d is located between M and the stack free pointer
  b = M.stack->free.address > (void *) q->_mp_d
      && (void *) q->_mp_d >= M.address;
// The stack free pointer is moved back to M : q->_mp_d is freed
  ba0_restore (&M);
  b = b && M.stack->free.address == M.address;
  ba0_terminate (ba0_init_level);
  return b ? 0 : 1;
}
```

4.2.5.2 The two stacks scheme

It is illustrated by the following example. It relies on the fact that there exists two general purpose stacks which can alternatively become the current stack (the one in which dynamic allocation is implicitly performed) by calling `ba0_push_another_stack`. The two stacks scheme works recursively.

```
#include "ba0.h"

// the field _mp_d of result points towards the current stack (say) main
static void
big_number (
    ba0_mpz_t result)
{
  struct ba0_mark M;
  ba0_mpz_t p;
// the second stack becomes the current stack
  ba0_push_another_stack ();
  ba0_record (&M);
// intermediate computations in the second stack
  ba0_mpz_init_set_si (p, 421);
  ba0_mpz_pow_ui (p, p, 30);
// the main stack becomes again the current stack
  ba0_pull_stack ();
// the field _mp_d of result is in main stack
  ba0_mpz_set (result, p);
// all the memory allocated in second stack is freed
  ba0_restore (&M);
}

int
main (
    void)
{
  ba0_mpz_t bignum;
  struct ba0_mark M;
  bool b;

  ba0_restart (0, 0);
  ba0_record (&M);
  ba0_mpz_init (bignum);
  big_number (bignum);
  b = M.stack->free.address >= (void *) (bignum->_mp_d + bignum->_mp_alloc);
  ba0_terminate (ba0_init_level);
  return b ? 0 : 1;
}
```

4.2.5.3 The garbage function

`void ba0_garbage (char *s, struct ba0_mark *M, ...)` [Function]

The front-end function for garbage collection. This function belongs to `ba0_garbage.c`. It is run by 77 tests, including 25 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

For some rare functions, the two stacks mechanism is difficult to apply and the garbage function is used. This function is also needed when raising exceptions returning values.

The function is much inspired from Jean–Charles Faugère's garbage collection mechanism. Here are the rules:

- The data structure lies in the current stack, which must be `ba0_main_stack` or `ba0_second_stack`.
- All dynamically allocated subareas stand in the same stack as the data structure (apart possibly stacks not concerned by garbage collection such as `ba0_quiet_stack`).

At the beginning of a function, one stores the value of the free pointer of the current stack in a mark M. At any step of the computation, one may call the `ba0_garbage` function with parameters M and the addresses of the variables that we want to keep (here R)

The function moves all the dynamically allocated subareas of the variables which lie in the range between M and the actual value of the free pointer. It pushes them close to M (it removes useless holes between the areas) and then updates the free pointer. The garbaged variables may involve subareas outside the range: these subareas are not moved. Different variables may share subareas: these subareas keep being shared, they are not duplicated.

Here is an example.

```
#include "ba0.h"

int
main (
    void)
{
  struct ba0_mark M;
  ba0_mpz_t p;
  char buffer[128];
  bool b;
// In this example, q and T are processed by the gargabe collector
// Thus these variables MUST be pointers and not structures in the
// process stack
  ba0__mpq_struct *q;
  struct ba0_tableof_mpz *T;

  ba0_restart (0, 0);
  ba0_mpz_init_set_si (p, 888);
// Mark M is set after p
  ba0_record (&M);
// Wasted memory
  ba0_new_mpz ();
// q is assigned some rational number
  q = ba0_new_mpq ();
  ba0_mpq_set_si_si (q, 22, 7);
// T is assigned a table of two numbers.
// The two entries are shared dynamically allocated areas
  T = (struct ba0_tableof_mpz *) ba0_new_table ();
  ba0_sscanf2 ("[421, 317]", "%t[%z]", T);
  T->tab[0] = T->tab[1];
// T is resized to get a third entry. The former field T->tab is lost
  ba0_realloc2_table ((struct ba0_table *) T, 3,
      (ba0_new_function *) & ba0_new_mpz);
  ba0_mpz_set_si (T->tab[2], 512);
  T->size = 3;
// Any area which is not part of q or of T is recovered.
// T and q are moved just above M. Formerly shared areas keep being shared.
// The order in memory is preserved: q, initialized first, is located at M.
```

```
  ba0_garbage ("%q%t[%z]", &M, &q, &T);
  b = T->tab[0] == T->tab[1] && M.address == (void *) q;
  ba0_sprintf (buffer, "%z, %q, %t[%z]", p, q, T);
  b = b && strcmp (buffer, "888, 22/7, [317, 317, 512]") == 0;
  ba0_terminate (ba0_init_level);
  return b ? 0 : 1;
}
```

The garbage collection algorithm is the following.

- A function runs over all the subareas of the variables given as parameters to `ba0_garbage` (parameters in the dots of the function prototype) and records their (old) addresses and their sizes (the `garbage1` fields in `ba0_format` data structures serve this purpose) in an array of data structures of type `ba0_gc_info`. A `ba0_gc_info` data structure is created only for the subareas which lie in the range between M and the free pointer.
- A second array is created with the addresses of the `ba0_gc_info` data structures just created and sorted by increasing order: one compares the (old) addresses of the subareas described by the `ba0_gc_info` data structures.
- The subareas are copied as close to `M` as possible (compression). The new addresses of the subareas are recorded in the `ba0_gc_info` data structures. The algorithm runs over the sorted second array created at the previous step. Thus, the order of the subareas in the memory is not modified and shared subareas (easily recognized) are not duplicated.
- A function runs over all the subareas of the variables given as parameters to the `ba0_garbage` function and updates the internal pointers between subareas. The subareas which do not lie in the range between M and the free pointer are of course updated also (the `garbage2` fields in `ba0_format` data structures serve this purpose). Note: since the pointers are incorrect, the array of `ba0_gc_info` data structures created at the first step is used to follow the links.

Description of type `ba0_garbage_code`. This data type is passed as an argument to `ba0_garbage1_function` and `ba0_garbage2_function` functions when applying the garbage collector to a data structure. This type declaration belongs to `ba0_common.h`.

```
enum ba0_garbage_code
{
// the data structure stands alone
  ba0_isolated,
// the data structure is a field of a bigger data structure
  ba0_embedded
};
```

Description of type `ba0_garbage1_function`. This type declaration belongs to `ba0_common.h`.

```
typedef ba0_int_p ba0_garbage1_function (
    void *,
    enum ba0_garbage_code);
```

Description of type `ba0_garbage2_function`. This type declaration belongs to `ba0_common.h`.

```
typedef void *ba0_garbage2_function (
    void *,
    enum ba0_garbage_code);
```

Description of type ba0_gc_info. Structures of this data type are created by ba0_garbage1_function and read by ba0_garbage2_function functions. They permit to move data structures involving pointers. This type declaration belongs to ba0_garbage.h.

```
struct ba0_gc_info
{
// the index of the stack cell which contains the garbaged data structure
  ba0_int_p old_index_in_cells;
// the old address of the data structure
  void *old_addr;
  union
  {
// the size in bytes of the data structure (set at garbage1 pass)
    unsigned ba0_int_p size;
// the new address of the data structure (set at garbage2 pass)
    void *new_addr;
  } u;
// a readonly string describing the structure for debugging purposes
  char *text;
};
```

4.3 Containers

4.3.1 Overview

The library provides four generic container shapes:

- Lists: linked, convenient for incremental construction.
- Tables: resizable one-dimensional sequences.
- Matrices: rectangular two-dimensional sequences.
- Values: pairs associating variables (to be defined in bav) to values

For each shape, ba0 provides specialised variants/implementations for the supported element types. A variant determines how elements are stored, printed by ba0_printf, read by ba0_scanf, copied by ba0_copy, and moved/preserved by ba0_garbage. Three specialised implementations are supported:

1. machine integers: the main data type is ba0_int_p; the data type ba0_bool is an alias for it. Lists, tables, matrices and values of ba0_int_p are handled by ba0_printf, ba0_scanf, ba0_copy and ba0_garbage by means of the formats %l[%d] (lists), %t[%d] (tables), %value(%d) (values) and %m[%d].
2. floating point numbers: the data type is double. Lists, tables, matrices and values of double are handled by means of the formats %l[%le] (lists), %t[%le] (tables), %value(%le) (values) and %m[%le].
3. pointers to data structures allocated outside the containers: this generic data type should be used by any new code, implementing new data structures. An example is given by the GMP rational numbers, which are denoted by the %q format. Lists, tables, matrices and values of GMP rational numbers are handled by means of the formats %l[%q] (lists), %t[%q] (tables), %value(%q) (values) and %m[%q].

4.4 Formats

Description of type `ba0_typeof_format_component`. This data type is a subtype of `ba0_format_component`. This type declaration belongs to `ba0_format.h`.

```
enum ba0_typeof_format_component
{
  ba0_atomic_format_component,
  ba0_list_format_component,
  ba0_table_format_component,
  ba0_value_format_component,
  ba0_point_format_component,
  ba0_matrix_format_component
};
```

Description of type `ba0_composite_format_component`. This type declaration belongs to `ba0_format.h`.

```
struct ba0_composite_format_component
{
  struct ba0_format *nested_component;
  char po;                      // opening parenthesis
  char pf;                      // closing parenthesis
};
```

Description of type `ba0_atomic_format_component`. This type declaration belongs to `ba0_format.h`.

```
struct ba0_atomic_format_component
{
  ba0_int_p sizelt;
  ba0_scanf_function *scanf;
  ba0_printf_function *printf;
  ba0_garbage1_function *garbage1;
  ba0_garbage2_function *garbage2;
  ba0_copy_function *copy;
};
```

Description of type `ba0_format_component`. This data type is a subtype of `ba0_format`. An element of this type describes a `%something` substring of a format. Such substrings may have parameters. Examples are: `%t[%v]` (tables of variables), `%l[%t[%z]]` (lists of tables of big integers). This type declaration belongs to `ba0_format.h`.

```
struct ba0_format_component
{
// the type of the format component
  enum ba0_typeof_format_component type;
  struct ba0_composite_format_component composite;
  struct ba0_atomic_format_component atomic;
};
```

Description of type `ba0_format`. This data type describes a format, which is a string that can be passed to functions such as `ba0_scanf`, `ba0_printf`, `ba0_copy` or `ba0_garbage`. Examples are: `"poly = %Az\n"` or `"%t[%v], %qi"`.

The format components `%something` which occur in such strings are preprocessed.

Each format is stored in a hash table of formats, stored in the format stack whenever it is encountered. Note that the addresses of the strings are hashed, not the strings themselves. This type declaration belongs to `ba0_format.h`.

```
struct ba0_format
{
```

```
// the string
  char *text;
// an array of pointers towards the format components of the string
  struct ba0_tableof_format_component component;
};
```

Description of type `ba0_scanf_function`. This type declaration belongs to `ba0_common.h`.

```
typedef void *ba0_scanf_function (
    void *);
```

Description of type `ba0_printf_function`. This type declaration belongs to `ba0_common.h`.

```
typedef void ba0_printf_function (
    void *);
```

Description of type `ba0_copy_function`. This type declaration belongs to `ba0_common.h`.

```
typedef void *ba0_copy_function (
    void *);
```

`void ba0_initialize_format (void)` [Function]

Initialize the table of the atomic formats and the H-table of formats. This function belongs to `ba0_format.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`struct ba0_format * ba0_get_format (char *`*`s`*`)` [Function]

Transform the string *s* to a format and return it. There are restrictions on the strings. In particular, parentheses must be paired. The format corresponding to a given string is only computed once. Exception `BA0_ERROOM` is raised if the H-table is full. This function belongs to `ba0_format.c`. It is run by 533 tests, including 43 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_define_format (char *`*`id`*`, ba0_scanf_function *`*`s`*`,` [Function]
`ba0_printf_function *`*`p`*`, ba0_garbage1_function *`*`g1`*`,`
`ba0_garbage2_function *`*`g2`*`, ba0_copy_function *`*`c`*`)`

Create a new atomic format, identified by the special character `%` followed by the string *id*. The five other parameters are pointers to functions which permit to perform input, output, garbage collection and copy.

Some pointers may be zero. In this case, the function is taken to be the empty function. Some pointers may be minus one. In this case, the function raises the `BA0_ERRNYP` exception (not yet implemented) when called.

Some exceptions may be raised: `BA0_ERRSYN` (syntax error), `BA0_ERRALG` (runtime error, in the case of a twice defined format). This function belongs to `ba0_format.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_define_format_with_sizelt (char *`*`id`*`, ba0_int_p` [Function]
`sizelt`*`, ba0_scanf_function *`*`s`*`, ba0_printf_function *`*`p`*`,`
`ba0_garbage1_function *`*`g1`*`, ba0_garbage2_function *`*`g2`*`,`
`ba0_copy_function *`*`c`*`)`

Variant of the above function in the case the object identified by the format is a pointer to some data structure. The parameter *sizelt* may then be -1 or the size of

the data structure the pointer points to. In the first case, arrays of data structures are forbidden. In the second case, they are allowed.

The previous function is a particular case of this one with $sizelt = -1$. This function belongs to `ba0_format.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_is_garbaged_format_component (struct ba0_format_component *component)` [Function]

This function belongs to `ba0_format.c`. It is run by 20 tests, including 10 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`bool ba0_is_int_p_format_component (struct ba0_format_component *component)` [Function]

This function belongs to `ba0_format.c`. It is run by 530 tests, including 42 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_has_int_p_format_component (struct ba0_format *f)` [Function]

This function belongs to `ba0_format.c`. It is run by 529 tests, including 41 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_is_double_format_component (struct ba0_format_component *component)` [Function]

This function belongs to `ba0_format.c`. It is run by 530 tests, including 42 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_has_double_format_component (struct ba0_format *f)` [Function]

This function belongs to `ba0_format.c`. It is run by 525 tests, including 37 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`struct ba0_format_component * ba0_nested_format_component (struct ba0_format_component *component)` [Function]

This function belongs to `ba0_format.c`. It is run by 140 tests, including 18 in the `ba0` library and 25 in the `DifferentialAlgebra` package.

4.5 An example of a new data structure definition

The following example shows how to create a new data structure and its corresponding format.

```
#include "ba0.h"

typedef struct my_complex       /* An awkward implementation! */
{
  double *realpart;             /* This a pointer to a double */
  double imagpart;              /* This is a double */
} *my_complex;

/*
 * default constructor
 */
static void
init_my_complex (
    my_complex z)
```

```
{
  z->realpart = ba0_new_double ();
}

/*
 * the new function
 */
static my_complex
new_my_complex (
    void)
{
  my_complex z;
  z = (my_complex) ba0_alloc (sizeof (struct my_complex));
  init_my_complex (z);
  return z;
}

/*
 * the assignment function
 */
static void
set_my_complex (
    my_complex z1,
    my_complex z2)
{
  *z1->realpart = *z2->realpart;
  z1->imagpart = z2->imagpart;
}

/*
 * a basic parser
 */
static void *
scanf_my_complex (
    void *x)
{
  my_complex z;
/*
 * Every scanf function receives either zero or a pointer to an initialized
 * data structure. In the first case, an area must be allocated.
 */
  if (x == (void *) 0)
    z = new_my_complex ();
  else
    z = (my_complex) x;
/*
 * The lexical analyzer is used for parsing my_complex
 */
  ba0_scanf ("%le", z->realpart);
  ba0_get_token_analex ();
  if (ba0_sign_token_analex ("+") || ba0_sign_token_analex ("-"))
    {
      double posneg = ba0_sign_token_analex ("+") ? 1. : 0.;
      ba0_get_token_analex ();
      ba0_scanf ("i*%le", &z->imagpart);
      z->imagpart *= posneg;
    }
  else
```

```
    {
      ba0_unget_token_analex (1);
      z->imagpart = 0.;
    }
/*
 * The pointer to the data structure must be returned
 */
  return z;
}

/*
 * The printing function
 */
static void
printf_my_complex (
    void *x)
{
  my_complex z = (my_complex) x;
  if (z->imagpart == 0.)
    ba0_printf ("%le", z->realpart);
  else if (z->imagpart < 0.)
    {
      double mz = -z->imagpart;
      ba0_printf ("%le - i * %le", z->realpart, &mz);
    }
  else
    ba0_printf ("%le + i * %le", z->realpart, &z->imagpart);
}

static char struct_my_complex[] = "struct my_complex";

static ba0_int_p
garbage1_my_complex (
    void *x,
    enum ba0_garbage_code code)
{
  my_complex z = (my_complex) x;
  ba0_int_p n;
/*
 * Isolated structures must be moved separately by the garbage collector.
 * Embedded structures are moved together with their embedding structure.
 */
  n = 0;
  if (code == ba0_isolated)
    {
      ba0_new_gc_info (z, sizeof (struct my_complex), struct_my_complex);
      n += 1;
    }
  n += ba0_garbage1 ("%le", z->realpart, ba0_isolated); /* needed */
  n += ba0_garbage1 ("%le", &z->imagpart, ba0_embedded);        /* pointless */
  return n;
}

static void *
garbage2_my_complex (
    void *x,
    enum ba0_garbage_code code)
{
```

```
  my_complex z;
/*
 * The string provided as second parameter to ba0_new_addr_gc_info
 * is only useful for debugging. It must be exactly the same as that
 * provided to ba0_new_gc_info by the garbage1 function.
 */
  if (code == ba0_isolated)
    z = (my_complex) ba0_new_addr_gc_info (x, struct_my_complex);
  else
    z = (my_complex) x;
  z->realpart = (double *) ba0_garbage2 ("%le", z->realpart, ba0_isolated);
  ba0_garbage2 ("%le", &z->imagpart, ba0_embedded);     /* pointless */
  return z;
}

/*
 * the copy function
 */
static void *
copy_my_complex (
    void *x)
{
  my_complex z;
  z = new_my_complex ();
  set_my_complex (z, (my_complex) x);
  return z;
}

int
main (
    void)
{
  struct ba0_table *A;
  struct my_complex pi, **tab;
  struct ba0_mark M;
  bool b;

  ba0_restart (0, 0);
/*
 * The new format %my_complex is defined
 */
  ba0_define_format_with_sizelt
      ("my_complex", sizeof (struct my_complex),
      &scanf_my_complex,
      &printf_my_complex,
      &garbage1_my_complex, &garbage2_my_complex, &copy_my_complex);
/*
 * Read one my_complex in pi. The variable must be initialized before parsing.
 */
  init_my_complex (&pi);
  ba0_sscanf2 ("3.14", "%my_complex", &pi);
/*
 * Set a mark M and waste 10 bytes
 */
  ba0_record (&M);
  ba0_alloc (10);
/*
 * Read a table of my_complex. Garbage collection recovers the wasted bytes
```

```
 */
  A = ba0_new_table ();
  ba0_sscanf2 ("[3. + i*5e-1, -7.e0]", "%t[%my_complex]", A);
  ba0_garbage ("%t[%my_complex]", &M, &A);
/*
 * Check
 */
  ba0_printf ("pi = %my_complex, A = %t[%my_complex]\n", &pi, A);
  tab = (my_complex *) A->tab;
  b = A->size == 2 &&
      *(tab[0]->realpart) == 3. && tab[0]->imagpart == 0.5 &&
      *(tab[1]->realpart) == -7. && tab[1]->imagpart == 0.;

  ba0_terminate (ba0_init_level);
  return b ? 0 : 1;
}
```

4.6 Exceptions

The memory management mechanism allows a simple and efficient exception mechanism based on `setjmp`/`longjmp`. Exceptions are identified by strings, which are stored in global variables. They are carried out by the macros `BA0_TRY`, `BA0_CATCH` and `BA0_ENDTRY`.

4.6.1 Basic exceptions

```
#include "ba0.h"

static double
invert_double (
    double z)
{
  struct ba0_mark M;

  ba0_record (&M);
// simulate some temporary memory allocation
  ba0_alloc (10);
  if (z == 0.)
    BA0_RAISE_EXCEPTION (BA0_ERRIVZ);
// the temporary area is freed only if no exception was raised
  ba0_restore (&M);
  return 1. / z;
}

int
main (
    void)
{
  struct ba0_tableof_double T;
  struct ba0_mark M;
  bool b = true;

  ba0_restart (0, 0);
  ba0_init_tableof_double (&T);
  ba0_sscanf2 ("[3., 0., -5]", "%t[%le]", &T);
  ba0_record (&M);
  for (int i = 0; i < T.size; i++)
    {
      BA0_TRY
```

```
      {
        double d;
        d = invert_double (T.tab[i]);
        ba0_printf ("the inverse of %le is: %le\n", &T.tab[i], &d);
// Check that the call is successful for i = 0 or 2
        b = b && (i == 0 || i == 2);
      }
      BA0_CATCH
      {
        if (ba0_global.exception.raised != BA0_ERRIVZ)
          BA0_RE_RAISE_EXCEPTION;
        ba0_printf ("%le has no inverse\n", &T.tab[i]);
// Check that exception BA0_ERRIVZ was raised for i == 1
        b = b && (i == 1);
      }
      BA0_ENDTRY;
    }
// Check that no memory was lost, even at loop i == 1
  b = b && M.address == M.stack->free.address;
  ba0_terminate (ba0_init_level);
  return b ? 0 : 1;
}
```

4.6.2 Exceptions with parameters

```
#include "ba0.h"

static double
invert_double (
    double z,
    char **mesg)
{
  struct ba0_mark M;

  ba0_record (&M);
// simulate some temporary memory allocation
  ba0_alloc (10);
  if (z == 0.)
    {
// the string is allocated in the main stack
      *mesg = ba0_strdup ("zero is not invertible!");
      BA0_RAISE_EXCEPTION2 (BA0_ERRIVZ, "%s", mesg);
    }
  else if (ba0_isnan (z))
    {
      ba0_push_another_stack ();
// the second stack becomes the current stack
// the string is allocated in the second stack
      *mesg = ba0_strdup ("nan is not invertible!");
      BA0_RAISE_EXCEPTION2 (BA0_ERRIVZ, "%s", mesg);
    }
// the temporary area is freed only if no exception was raised
  ba0_restore (&M);
  return 1. / z;
}

int
main (
    void)
```

```
{
  struct ba0_tableof_double T;
  struct ba0_stack *H;
  struct ba0_mark M;
  bool b = true;

  ba0_restart (0, 0);
  ba0_init_tableof_double (&T);
  ba0_sscanf2 ("[3., 0., nan, -5]", "%t[%le]", &T);
  H = ba0_current_stack ();
  ba0_record (&M);
  for (int i = 0; i < T.size; i++)
    {
      char *mesg;
      BA0_TRY
      {
        double d;
        d = invert_double (T.tab[i], &mesg);
        ba0_printf ("the inverse of %le is: %le\n", &T.tab[i], &d);
// Check that the call is successful for i = 0 or 3
        b = b && (i == 0 || i == 3);
      }
      BA0_CATCH
      {
        if (ba0_global.exception.raised != BA0_ERRIVZ)
          BA0_RE_RAISE_EXCEPTION;
        ba0_printf ("%s\n", mesg);
// Check that exception BA0_ERRIVZ was raised for i == 1 or 2
//   and that mesg is in the main stack
        b = b && (i == 1 || i == 2) && ba0_in_stack (mesg, H);
      }
      BA0_ENDTRY;
    }
// Check that the current stack is the main one
  b = b && H == ba0_current_stack ();
  ba0_terminate (ba0_init_level);
  return b ? 0 : 1;
}
```

4.6.3 Data structures

When an exception catching point is set, some information is recorded in the global variable `ba0_global.exception`. Some is recorded in a local variable created by `BA0_TRY`. Here are some details.

Description of type `ba0_exception`. This data type implements the data stored in the entries of `ba0_global.exception.stack`. These entries are filled when an exception catching point is set. They are used for restoring values when an exception is caught. This type declaration belongs to `ba0_exception.h`.

```
struct ba0_exception
{
  struct ba0_jmp_buf_struct jmp_b;      // for setjmp/longjmp
// a pointer to the local variable __code__ set by BA0_TRY
  struct ba0_exception_code *code;
};
```

Description of type `ba0_exception_code`. The data type for local variables `__code__` set by `BA0_TRY`. This type declaration belongs to `ba0_exception.h`.

```
struct ba0_exception_code
{
// a copy of ba0_global.exception.stack.size for debugging purpose
  ba0_int_p exception_stack_size;
  bool cancelled;               // set to true by BA0_ENDTRY
  int jmp_code;                 // the returned value of setjmp
// values recorded when an exception catching point is set
  struct ba0_mark main;         // the free pointer of the main stack
  struct ba0_mark second;       // the one of the second stack
  ba0_int_p stack_of_stacks_size;       // the field size of the stack of stacks
// values of extra variables to be saved/restored
// the pointers to the variables are in ba0_global.exception.extra_stack
  struct
  {
    ba0_int_p tab[BA0_SIZE_EXCEPTION_EXTRA_STACK];
    ba0_int_p size;
  } extra_stack;
};
```

4.6.4 The exception extra stack

It permits to make some further variables saved and restored by the exception mechanism. An example is provided by the following piece of code, borrowed from the `bav` library. The content of `bav_global.R.ord_stack.size` is saved and restored by the exception mechanism. The content of `bav_global.R.ords.size` is saved by the exception mechanism. It is restored by calling the function `bav_R_restore_ords_size`.

```
ba0_push_exception_extra_stack
    (&bav_global.R.ord_stack.size, (void (*)(ba0_int_p)) 0);
ba0_push_exception_extra_stack
    (&bav_global.R.ords.size, &bav_R_restore_ords_size);
```

4.6.5 Exceptions primitives

`void ba0_reset_exception_extra_stack (void)` [Function]

Empty the exception extra stack. This function belongs to `ba0_exception.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_push_exception_extra_stack (ba0_int_p *`*`pointer`*`,` [Function]
`void (*`*`restore`*`) (ba0_int_p))`

Stack p and r in the exception extra stack. This function belongs to `ba0_exception.c`. It is run by 494 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_pull_exception_extra_stack (void)` [Function]

Undo the effect of the last call to `ba0_push_exception_extra_stack`. This function belongs to `ba0_exception.c`. It is not run by any test.

`BA0_CERR (`*`msg`*`)` *char* msg* [Macro]

Print the error message on the standard error and exits with code 1.

`void ba0_reset_exception (void)` [Function]
Empty the stack of exception catching points. This function belongs to `ba0_exception.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`BA0_TRY` [Macro]
Declare a local variable `__code__` of type `struct ba0_exception_code`, set an exception catching point by calling `ba0_push_exception` and `ba0_setjmp`.

`BA0_CATCH` [Macro]
An `else` statement, working with `BA0_TRY`.

`BA0_ENDTRY` [Macro]
Remove the local variable `__code__` and cancel the exception by calling `ba0_pull_exception`.

`BA0_CANCEL_EXCEPTION` [Macro]
Cancel the exception catching point by calling `ba0_pull_exception`. This macro is sometimes useful to cancel an exception catching point between `BA0_TRY` and `BA0_CATCH`.

`void ba0_push_exception (struct ba0_exception_code *`*`code`*`)` [Function]
Set an exception catching point. This function is called by `BA0_TRY` and stores some information in the local variable *code* and some other information in the exception stack stored in `ba0_global.exception`. The exception log is reset. This function belongs to `ba0_exception.c`. It is run by 543 tests, including 49 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_pull_exception (struct ba0_exception_code *`*`code`*`)` [Function]
Undo the last call to `BA0_PUSH_EXCEPTION`. Check for debugging purpose that the value of the exception stack pointer is correct. This function belongs to `ba0_exception.c`. It is run by 539 tests, including 49 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_exception_is_set (struct ba0_exception_code *`*`code`*`)` [Function]
Return `true` if the exception catching point has just been set and `false` otherwise. This function belongs to `ba0_exception.c`. It is run by 539 tests, including 49 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_exception_is_raised (struct ba0_exception_code *`*`code`*`)` [Function]
Return `true` if the exception catching point is catching an exception and `false` otherwise. This function belongs to `ba0_exception.c`. It is not run by any test.

`BA0_RAISE_EXCEPTION (`*`msg`*`)` *char* msg* [Macro]
Raise the exception identified by *msg*.

`BA0_RAISE_PARSER_EXCEPTION (`*`msg`*`)` *char* msg* [Macro]
Call `ba0_write_context_analex` and raise the exception identified by *msg*.

`BA0_RAISE_EXCEPTION2 (msg, f, o) char* msg, char* f, void** o` [Macro]

Raise the exception identified by *msg* but preserve from deletion the value of the variable **o* the type of which is described by the format *f*. The function ensures that the value of **o* lies in the stack which was current when the exception catching point was set (technically, either `ba0_garbage` or `ba0_copy` is applied to **o* depending on the current stacks).

`void ba0_raise_exception (char *file, int line, char *raised)` [Function]

Raise exception *raised* from file *file* at line *line*. These data are queued in the exception log of `ba0_global.exception`. Exceptions `BA0_ERRALG` or `BA0_ERRNYP` cause an extra error message of `stderr`. This function belongs to `ba0_exception.c`. It is run by 115 tests, including 11 in the `ba0` library and 21 in the `DifferentialAlgebra` package.

`void ba0_raise_exception2 (char *file, int line, char *raised, char *f, void **o)` [Function]

Variant of `ba0_raise_exception`. See the macro `BA0_RAISE_EXCEPTION2`. This function belongs to `ba0_exception.c`. It is run by 35 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.7 Input and output

4.7.1 Basic output

The global variable `ba0_output_line_length` contains the maximal number of characters that should be printed on a single line (i.e. without any newline character) when the output is redirected to a file. A zero value means no limit. Its default value is 80.

Description of type `ba0_typeof_device`. This data type permits to specify the device from/to which the input is read of the output is written. This type declaration belongs to `ba0_basic_io.h`.

```
enum ba0_typeof_device
{
  ba0_string_device,
  ba0_file_device,
  ba0_counter_device
};
```

Description of type `ba0_output_device`. This data type the device to which output is written. This type declaration belongs to `ba0_basic_io.h`.

```
struct ba0_output_device
{
  enum ba0_typeof_device vers;
// if vers is equal to ba0_file_device
  FILE *file_flux;
// if vers is equal to ba0_string_device
  char *string_flux;
// index in string_flux
  ba0_int_p indice;
// if vers is equal to ba0_counter_device (number of output characters)
  ba0_int_p counter;
};
```

`void ba0_reset_output (void)` [Function]
Reset the output to its default state (i.e. to `stdout`) and `ba0_output_line_length` to its default value. This function belongs to `ba0_basic_io.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_output_FILE (FILE *f)` [Function]
Make all output functions write in the file *f*. This function belongs to `ba0_basic_io.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_output_string (char *s)` [Function]
Make all output functions write in *s*, which must points to a large enough area. This function belongs to `ba0_basic_io.c`. It is run by 528 tests, including 39 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_output_counter (void)` [Function]
Make all output functions write on a counter (automatically reset to zero). This function may be used to determine the length of an expression before storing it in a string. This function belongs to `ba0_basic_io.c`. It is run by 502 tests, including 15 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`ba0_int_p ba0_output_counter (void)` [Function]
Return the value of the counter. This function belongs to `ba0_basic_io.c`. It is run by 502 tests, including 15 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_record_output (void)` [Function]
Push the current value of the output in the stack `ba0_global.basic_io.output_stack`. Exception `BA0_ERRSOV` is raised if the saving stack is full. This function belongs to `ba0_basic_io.c`. It is run by 528 tests, including 39 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_restore_output (void)` [Function]
Pull the top element from the stack `ba0_global.basic_io.output_stack`. Exception `BA0_ERRSOV` is raised if the saving stack is empty. This function belongs to `ba0_basic_io.c`. It is run by 528 tests, including 39 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_put_char (char c)` [Function]
Send *c* to the output. If the output is a file then newlines may be inserted (see above). This function belongs to `ba0_basic_io.c`. It is run by 506 tests, including 37 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_put_string (void *s)` [Function]
Send the string pointed to by *s* to the output. If the output is a file then newlines may be inserted (see above). The string is however not split. This function belongs to `ba0_basic_io.c`. It is run by 530 tests, including 41 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_put_int_p (ba0_int_p n)` [Function]
Print n in base 10 and send the string to the output. This function belongs to `ba0_basic_io.c`. It is run by 320 tests, including 7 in the `ba0` library and 53 in the `DifferentialAlgebra` package.

`void ba0_put_hexint_p (ba0_int_p n)` [Function]
Print n in base 16 and send the string to the output. This function belongs to `ba0_basic_io.c`. It is not run by any test.

4.7.2 Basic input

Description of type `ba0_input_device`. This data type the device from which input is read. This type declaration belongs to `ba0_basic_io.h`.

```
struct ba0_input_device
{
  enum ba0_typeof_device from;
// if from == ba0_file_device
  FILE *file_flux;
// if from == ba0_string_device
  char *string_flux;
  ba0_int_p indice;
};
```

`void ba0_reset_input (void)` [Function]
Reset the input to its initial state (i.e. from `stdin`). This function belongs to `ba0_basic_io.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_input_FILE (FILE *f)` [Function]
Make all input functions read in the file f. This function belongs to `ba0_basic_io.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_input_string (char *s)` [Function]
Make all input functions read in the string s. This function belongs to `ba0_basic_io.c`. It is run by 317 tests, including 20 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_record_input (void)` [Function]
Push the current value of the input in the stack `ba0_global.basic_io.input_stack`. Exception `BA0_ERRSOV` is raised if the saving stack is full. This function belongs to `ba0_basic_io.c`. It is run by 317 tests, including 17 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_restore_input (void)` [Function]
Pull the top element from the stack `ba0_global.basic_io.input_stack`. Exception `BA0_ERRSOV` is raised if the saving stack is empty. This function belongs to `ba0_basic_io.c`. It is run by 317 tests, including 17 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_isatty_input (void)` [Function]
Return `true` if the input is a *tty*, `false` otherwise. This function belongs to `ba0_basic_io.c`. It is run by 8 tests, including 2 in the `ba0` library and 2 in the `DifferentialAlgebra` package.

`int ba0_get_char (void)` [Function]
Read a character on the input. Returns `EOF` when the end of the input stream is reached. Backslashes and all the spaces that follow them are skipped. This function belongs to `ba0_basic_io.c`. It is run by 531 tests, including 41 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_unget_char (int c)` [Function]
Replace *c* on the input. If the input is a string, this operation must not modify the stream (i.e. one cannot unget a character which was not read). Exception `BA0_ERRALG` is raised otherwise. This exception is also raised if too many characters are ungot. This function belongs to `ba0_basic_io.c`. It is run by 531 tests, including 41 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.7.3 The lexical analyzer

The lexical analyzer extracts *tokens* from the input. Tokens are stored in a special stack: `ba0_analex_stack`. Tokens have a type: a `ba0_integer_token` is a sequence of digits ; a `ba0_string_token` is either a sequence of letters, digits, underscores not starting by a digit, or a sequence of characters enclosed between double quotes ; a `ba0_sign_token` is any other single character. The special type `ba0_no_token` is used when initializing and when end of files are encountered. A fifo of tokens is implemented which allows to unget tokens. The maximum length for tokens is computed from the cell size of `ba0_analex_stack` (about one half).

4.7.3.1 The context string

`void ba0_write_context_analex (void)` [Function]
Print the current token in the global variable `ba0_global.analex.context`. The token is enclosed between `-->` and `<--`. Some previous tokens are printed before. If the input is not a *tty*, some next tokens are printed afterwards. The string `ba0_global.analex.context` can be used for printing helpful messages when an error occurs at parsing time. This function is called by `BA0_RAISE_PARSER_EXCEPTION`. This function belongs to `ba0_analex.c`. It is run by 8 tests, including 2 in the `ba0` library and 2 in the `DifferentialAlgebra` package.

`char * ba0_get_context_analex (void)` [Function]
Return the address of the global variable `ba0_global.analex.context`. This function belongs to `ba0_analex.c`. It is run by 2 tests, including 0 in the `ba0` library and 2 in the `DifferentialAlgebra` package.

4.7.3.2 Lexical analyzer setting

`void ba0_set_settings_analex (ba0_int_p nbtokens, char *quotes)` [Function]
Set to *nbtokens* the length of the fifo used by the lexical analyzer. This number is not allowed to be less than 10. Set to *quotes* a string containing the characters that can

be used for delimiting tokens by quotes. Zero value reset the fields to default values. This function must be called outside sequences of calls to the library. This function belongs to `ba0_analex.c`. It is run by 65 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_analex (ba0_int_p *nbtokens, char **quotes)` [Function]

Assign to **nbtokens* the length of the fifo used by the lexical analyzer. Assign to **quotes* the string containing the characters that can be used for delimiting tokens by quotes. Pointers are allowed to be zero. This function belongs to `ba0_analex.c`. It is run by 24 tests, including 4 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.7.3.3 The substitution dictionary

`void ba0_reset_subs_dict_analex (void)` [Function]

Reset the values of the fields `subs_dict`, `subs_keys` and `subs_vals` of `ba0_global.analex`. This function belongs to `ba0_analex.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_push_subs_dict_analex (struct ba0_dictionary_string *subs_dict, struct ba0_tableof_string *subs_keys, struct ba0_tableof_string *subs_vals)` [Function]

Assign *subs_dict*, *subs_keys* and *subs_vals* to the corresponding fields of `ba0_global.analex`. Exception `BA0_ERRSOV` is raised if these fields are nonzero (the fields somehow implement a stack with a single entry). Exception `BA0_ERRALG` is raised if the three arguments have inconsistent sizes.

Note that the substitution mechanism is still experimental: keys are restricted to tokens of type `ba0_string_token`; values are restricted to tokens of type `ba0_integer_token`. This function belongs to `ba0_analex.c`. It is run by 102 tests, including 1 in the `ba0` library and 11 in the `DifferentialAlgebra` package.

`void ba0_pull_subs_dict_analex (void)` [Function]

Reset the values of the fields `subs_dict`, `subs_keys` and `subs_vals` of `ba0_global.analex`. Exception `BA0_ERRSOV` is raised if these fields are zero. This function belongs to `ba0_analex.c`. It is run by 102 tests, including 1 in the `ba0` library and 11 in the `DifferentialAlgebra` package.

4.7.3.4 Lexical analyzer primitives

Description of type `ba0_typeof_token`. This data type permits to assign a type to a token. This type declaration belongs to `ba0_analex.h`.

```
enum ba0_typeof_token
{
// Special value
  ba0_no_token,
// The token is an integer starting with a digit (unsigned)
  ba0_integer_token,
// The token is a sign. Signs are single characters
  ba0_sign_token,
```

```
// The token is a string.
// Any sequence of characters delimited by quotes is a string
  ba0_string_token
};
```

Description of type `ba0_token`. This data type permits to store one token. This type declaration belongs to `ba0_analex.h`.

```
struct ba0_token
{
// The type of the token
  enum ba0_typeof_token type;
// At least one space was present before the first token character
  bool spaces_before;
// The value of the token, stored in the analex stack.
  char *value;
};
```

Description of type `ba0_analex_token_fifo`. This data type describes a FIFO of tokens implemented as a circular array. The only variables of this type are fields of `ba0_global`. This type declaration belongs to `ba0_analex.h`.

```
struct ba0_analex_token_fifo
{
// The array of token containing the FIFO.
// Its length is stored in ba0_initialized_global.analex.nb_tokens
  struct ba0_token *fifo;
// The index of the first token in the FIFO i.e the current token
  ba0_int_p first;
// The index of the last token present in the FIFO
// It is usually equal to first unless a token has been "ungot"
  ba0_int_p last;
// The number of calls to ba0_get_token_analex since the last reset
  ba0_int_p counter;
};
```

`void ba0_init_analex (void)` [Function]

Initialize the fields of `ba0_global` dedicated to the lexical analyzer. This function belongs to `ba0_analex.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_clear_analex (void)` [Function]

Free the resources allocated to the lexical analyzer. This function belongs to `ba0_analex.c`. It is run by 547 tests, including 53 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_analex (void)` [Function]

Reset the lexical analyzer. Recorded lexical analyzers are erased. the stack `ba0_global.stack.analex` is cleaned. The function `ba0_reset_input` is called. The string `ba0_global.analex.context` is cleared. The counter of calls to `ba0_get_token_analex` is reset to zero. The substitution dictionary is reset. This function belongs to `ba0_analex.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_record_analex (void)` [Function]

Record the value of the current lexical analyzer for future restoration. This function permits to functions such as `ba0_sscanf` to apply the lexical analyzer over a string and

restore afterwards the lexical analyzer as it was when they were called. Exception BA0_ERRSOV is raised if the saving variable is full. This function belongs to ba0_analex.c. It is run by 317 tests, including 17 in the ba0 library and 65 in the DifferentialAlgebra package.

void ba0_restore_analex (void) [Function]
Restore the value of the current lexical analyzer saved by the ba0_record_analex. Exception BA0_ERRSOV is raised if the saving variable is empty. This function belongs to ba0_analex.c. It is run by 317 tests, including 17 in the ba0 library and 65 in the DifferentialAlgebra package.

void ba0_set_analex_FILE (FILE *f) [Function]
Initialize the lexical analyzer. The input stream is now read in *f*. The counter of calls to ba0_get_token_analex is reset to zero. This function belongs to ba0_analex.c. It is run by 3 tests, including 0 in the ba0 library and 0 in the DifferentialAlgebra package.

void ba0_set_analex_string (char *s) [Function]
Initialize the lexical analyzer. The input stream is now read in *s*. The counter of calls to ba0_get_token_analex is reset to zero. This function belongs to ba0_analex.c. It is run by 317 tests, including 20 in the ba0 library and 65 in the DifferentialAlgebra package.

void ba0_get_token_analex (void) [Function]
Read a new token on the input stream. The counter of calls to ba0_get_token_analex is increased by 1. If the input stream is empty then the current token receives the type ba0_no_token. This function belongs to ba0_analex.c. It is run by 531 tests, including 41 in the ba0 library and 65 in the DifferentialAlgebra package.

ba0_int_p ba0_get_counter_analex (void) [Function]
Return the current value of the counter of calls to ba0_get_token_analex. This function belongs to ba0_analex.c. It is run by 291 tests, including 2 in the ba0 library and 55 in the DifferentialAlgebra package.

void ba0_unget_token_analex (ba0_int_p *n*) [Function]
Replace the *n* previous tokens on the input stream. The counter of calls to ba0_get_token_analex is decreased by *n*. This function belongs to ba0_analex.c. It is run by 531 tests, including 41 in the ba0 library and 65 in the DifferentialAlgebra package.

void ba0_unget_given_token_analex (char *value, enum ba0_typeof_token *type*, bool *spaces_before*) [Function]
The current token is erased. Its value is replaced by *value*, *type* and *spaces_before*. It will be read by the next call to ba0_get_token_analex. The string *value* is duplicated. The counter of calls to ba0_get_token_analex is decreased by 1. This function belongs to ba0_analex.c. It is run by 8 tests, including 4 in the ba0 library and 0 in the DifferentialAlgebra package.

`bool ba0_sign_token_analex (char *s)` [Function]
Return `true` is the current token has type `ba0_sign_token` and is equal to *s*. This function belongs to `ba0_analex.c`. It is run by 531 tests, including 41 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_spaces_before_token_analex (void)` [Function]
Return `true` if at least one space was skipped before reading the value of the current token. Comments are not considered as spaces. This function belongs to `ba0_analex.c`. It is run by 97 tests, including 13 in the `ba0` library and 14 in the `DifferentialAlgebra` package.

`enum ba0_typeof_token ba0_type_token_analex (void)` [Function]
Return the type of the current token after a possible replacement by the substitution dictionary. This function belongs to `ba0_analex.c`. It is run by 530 tests, including 40 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`char * ba0_value_token_analex (void)` [Function]
Return the value of the current token i.e. the string read on the input stream (whatever the type of the token) unless a replacement has been performed using the substitution dictionary. The memory used by the token is reused by the lexical analyzer after a number of calls to `ba0_get_token_analex` which depends on the length of the analyzer fifo. This function belongs to `ba0_analex.c`. It is run by 531 tests, including 41 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.7.4 High level input and output

Input is performed by reading through the lexical analyzer.

`void ba0_scanf (char *s, ...)` [Function]
Variant of `scanf` using formats. Many exception may be raised. Here is an example. The succession of tokens read on the format must match the ones read on the lexical analyzer. The function expects a list of strings and a table of blocks.

```
ba0_scanf ("ring (ders = %l[%s], ranking = %t[%b]);", &ls, &lb);
```

Classical trap: before calling `ba0_scanf`, the first token of the expression to be read must be the current token. After termination of `ba0_scanf`, the current token is the last token of the expression. This function belongs to `ba0_scanf.c`. It is run by 530 tests, including 40 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_scanf2 (char *s, ...)` [Function]
Variant of the above function, a bit more user friendly, which loads the first token of the expression to be read and then calls `ba0_scanf`. This function belongs to `ba0_scanf.c`. It is run by 506 tests, including 19 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_sscanf (char *buffer, char *s, ...)` [Function]
Read in *buffer*. Same remarks as `ba0_scanf`. This function belongs to `ba0_scanf.c`. It is run by 259 tests, including 17 in the `ba0` library and 64 in the `DifferentialAlgebra` package.

`void ba0_sscanf2 (char *buffer, char *s, ...)` [Function]
Read in *buffer*. Same remarks as `ba0_scanf2`. This function belongs to `ba0_scanf.c`. It is run by 259 tests, including 17 in the `ba0` library and 64 in the `DifferentialAlgebra` package.

`void ba0_fscanf (FILE *file, char *s, ...)` [Function]
Read in *file*. Same remarks as `ba0_scanf`. This function belongs to `ba0_scanf.c`. It is run by 3 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_fscanf2 (FILE *file, char *s, ...)` [Function]
Read in *file*. Same remarks as `ba0_scanf2`. This function belongs to `ba0_scanf.c`. It is run by 3 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_printf (char *s, ...)` [Function]
Variant of `printf`. This function belongs to `ba0_printf.c`. It is run by 530 tests, including 41 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_sprintf (char *buffer, char *s, ...)` [Function]
Variant of `sprintf`. Print in *buffer*. This function belongs to `ba0_printf.c`. It is run by 445 tests, including 36 in the `ba0` library and 17 in the `DifferentialAlgebra` package.

`void ba0_fprintf (FILE *file, char *s, ...)` [Function]
Variant of `fprintf`. Print in *file*. This function belongs to `ba0_printf.c`. It is run by 8 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`char * ba0_new_printf (char *s, ...)` [Function]
Print the expressions in a dynamically allocated (through `ba0_persistent_malloc`) string of the right length and returns it. The string still exists after executing `ba0_terminate`. This function belongs to `ba0_printf.c`. It is run by 224 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_scanf_printf (char *sf, char *pf, ...)` [Function]
Kind of a pipeline using formats. The objects corresponding to *pf* are printed in a buffer following the *pf* format. They are then read into the objects corresponding to *sf* using this format. Example. Assume P is a differential polynomial, u is a dependent variable and x a derivation. The following instruction sets P to $u_x^2 - 4u$.

```
ba0_scanf_printf ("%Az", "%v[%y]^2 - 4*%v", P, u, x, u);
```

This function belongs to `ba0_scanf_printf.c`. It is run by 155 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.8 Generic functions

The following data types permit to implement some generic functions.

Description of type `ba0_unary_function`. This type declaration belongs to `ba0_common.h`.

```
typedef void *ba0_unary_function (
```

```
    void *);
```

Description of type `ba0_cmp_function`. This type declaration belongs to `ba0_common.h`.

```
typedef bool ba0_cmp_function (
    void *,
    void *);
```

Description of type `ba0_cmp2_function`. This type declaration belongs to `ba0_common.h`.

```
typedef bool ba0_cmp2_function (
    void *,
    void *,
    void *);
```

Description of type `ba0_unary_predicate`. This type declaration belongs to `ba0_common.h`.

```
typedef bool ba0_unary_predicate (
    void *);
```

Description of type `ba0_new_function`. This type declaration belongs to `ba0_common.h`.

```
typedef void *ba0_new_function (
    void);
```

Description of type `ba0_init_function`. This type declaration belongs to `ba0_common.h`.

```
typedef void ba0_init_function (
    void *);
```

Description of type `ba0_unary_operation`. This type declaration belongs to `ba0_common.h`.

```
typedef void ba0_unary_operation (
    void *);
```

Description of type `ba0_binary_operation`. This type declaration belongs to `ba0_common.h`.

```
typedef void ba0_binary_operation (
    void *,
    void *);
```

Description of type `ba0_ternary_operation`. This type declaration belongs to `ba0_common.h`.

```
typedef void ba0_ternary_operation (
    void *,
    void *,
    void *);
```

4.9 Lists

4.9.1 Generic lists

Description of type `ba0_list`. This data structure implements simply linked lists of pointers to the list elements. The field `value` points to the field element. The field `next` points to the next list element. Assume that the format identifier for list elements is `%elt`. Then the format identifier for the list is `%l[%elt]`. This type declaration belongs to `ba0_list.h`.

```
struct ba0_list
{
```

```
  void *value;
  struct ba0_list *next;
};
```

`struct ba0_list * ba0_sort_list (struct ba0_list *L, ba0_cmp_function *f)` [Function]

Sort the list in place and returns its head. if $f(a,b) \Leftrightarrow a > b$ then the list is sorted by increasing order. This function belongs to `ba0_list.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`struct ba0_list * ba0_sort2_list (struct ba0_list *L, ba0_cmp2_function *f, void *data)` [Function]

ort the list in place and returns its head. if $f(a,b,data) \Leftrightarrow a > b$ then the list is sorted by increasing order. This function belongs to `ba0_list.c`. It is run by 103 tests, including 0 in the `ba0` library and 24 in the `DifferentialAlgebra` package.

`struct ba0_list * ba0_insert2_list (void *p, struct ba0_list *L, ba0_cmp2_function *f, void *data)` [Function]

If $f(a,b,data) = a > b$ and the list i sorted in decreasing order then p is inserted at the right place. This function belongs to `ba0_list.c`. It is run by 71 tests, including 0 in the `ba0` library and 13 in the `DifferentialAlgebra` package.

`struct ba0_list * ba0_cons_list (void *v, struct ba0_list *L)` [Function]

Insert v at the beginning of L. This function belongs to `ba0_list.c`. It is run by 526 tests, including 39 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`struct ba0_list * ba0_reverse_list (struct ba0_list *L)` [Function]

Return and returns L. In place function. This function belongs to `ba0_list.c`. It is run by 282 tests, including 0 in the `ba0` library and 39 in the `DifferentialAlgebra` package.

`ba0_int_p ba0_length_list (struct ba0_list *L)` [Function]

Return the number of elements of L. This function belongs to `ba0_list.c`. It is run by 528 tests, including 38 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.9.2 Lists of machine integers

Lists of `ba0_bool` are aliases of lists of machine integers.

Description of type `ba0_listof_int_p`. This type declaration belongs to `ba0_listof_int_p.h`.

```
struct ba0_listof_int_p
{
  ba0_int_p value;
  struct ba0_listof_int_p *next;
};
```

`struct ba0_listof_int_p * ba0_cons_listof_int_p (ba0_int_p value, struct ba0_listof_int_p *list)` [Function]

This function belongs to `ba0_listof_int_p.c`. It is run by 62 tests, including 6 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`ba0_int_p ba0_length_listof_int_p (struct ba0_listof_int_p *list)` [Function]

This function belongs to `ba0_listof_int_p.c`. It is run by 62 tests, including 6 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.9.3 Lists of floating point numbers

Description of type `ba0_listof_double`. This type declaration belongs to `ba0_listof_double.h`.

```
struct ba0_listof_double
{
  double value;
  struct ba0_listof_double *next;
};
```

`struct ba0_listof_double * ba0_cons_listof_double (double value, struct ba0_listof_double *list)` [Function]

This function belongs to `ba0_listof_double.c`. It is run by 15 tests, including 3 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`ba0_int_p ba0_length_listof_double (struct ba0_listof_double *list)` [Function]

This function belongs to `ba0_listof_double.c`. It is run by 15 tests, including 3 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.10 Tables

4.10.1 Generic tables

Description of type `ba0_table`. This data type implements resizable arrays of pointers. The field `tab` points to an array of pointers to the table elements. The field `alloc` gives the number of allocated elements. The field `size` gives the number of used elements. Assume that the format identifier for table elements is `%elt`. Then the format identifier for the table is `%t[%elt]`. This type declaration belongs to `ba0_table.h`.

```
struct ba0_table
{
  ba0_int_p alloc;
  ba0_int_p size;
  void **tab;
};
```

`void ba0_init_table (struct ba0_table *T)` [Function]

Initialize T (constructor). This function belongs to `ba0_table.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_table (struct ba0_table *T)` [Function]

Set to zero the number of used elements of T. This function belongs to `ba0_table.c`. It is run by 525 tests, including 35 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`struct ba0_table * ba0_new_table (void)` [Function]
Allocate a new `struct ba0_table` in the current stack, apply the constructor over it and return the result. This function belongs to `ba0_table.c`. It is run by 513 tests, including 23 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_table (struct ba0_table *T, enum ba0_garbage_code code)` [Function]
Return the size needed to perform a copy of T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `ba0_table.c`. It is run by 77 tests, including 3 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_realloc_table (struct ba0_table *T, ba0_int_p n)` [Function]
Reallocate the table if needed in such a way that the table can receive at least n elements. Formerly stored elements are kept. New entries are initialized to zero. This function belongs to `ba0_table.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_realloc2_table (struct ba0_table *T, ba0_int_p n, ba0_new_function *new_objet)` [Function]
Variant of the above function but initialize the new entries with the values returned by *new_objet*. This function belongs to `ba0_table.c`. It is run by 502 tests, including 12 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_re_malloc_table (struct ba0_table *T, ba0_int_p n)` [Function]
The entries of the table T are supposed to be allocated using `ba0_malloc`. Reallocate the table if needed (using `ba0_malloc`) in such a way that the table can receive at least n elements. Formerly stored elements are kept. This function belongs to `ba0_table.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_table (struct ba0_table *T, struct ba0_table *U)` [Function]
Copy U to T. This function belongs to `ba0_table.c`. It is run by 524 tests, including 34 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set2_table (struct ba0_table *T, struct ba0_table *U, ba0_new_function *newf, ba0_set_function *set)` [Function]
Copy U to T, duplicating each element of T. Entries of T and U are allowed to be the zero pointer. This function belongs to `ba0_table.c`. It is run by 39 tests, including 0 in the `ba0` library and 8 in the `DifferentialAlgebra` package.

`void ba0_delete_table (struct ba0_table *T, ba0_int_p i)` [Function]
Remove the ith element of T. This function belongs to `ba0_table.c`. It is run by 182 tests, including 0 in the `ba0` library and 33 in the `DifferentialAlgebra` package.

`void ba0_insert_table (struct ba0_table *T, ba0_int_p i, void *p)` [Function]
Insert p in T at index i. Exception `BA0_ERRALG` is raised if i is negative or greater than the number of used elements of T. This function belongs to `ba0_table.c`. It

is run by 8 tests, including 2 in the ba0 library and 2 in the DifferentialAlgebra package.

`bool ba0_member_table (void *p, struct ba0_table *T)` [Function]
Return `true` if x belongs to T else `false`. This function belongs to `ba0_table.c`. It is run by 251 tests, including 0 in the ba0 library and 40 in the DifferentialAlgebra package.

`bool ba0_member2_table (void *p, struct ba0_table *T, ba0_int_p *index)` [Function]
Return `true` if x belongs to T else `false`. In the first case, *index is assigned the index of x in T, provided that *index* is nonzero. This function belongs to `ba0_table.c`. It is run by 101 tests, including 0 in the ba0 library and 11 in the DifferentialAlgebra package.

`void ba0_set_table_list (struct ba0_table *T, struct ba0_list *L)` [Function]
Copy L to T. This function belongs to `ba0_table.c`. It is run by 517 tests, including 27 in the ba0 library and 65 in the DifferentialAlgebra package.

`void ba0_append_table_list (struct ba0_table *T, struct ba0_list *L)` [Function]
Append L to T. This function belongs to `ba0_table.c`. It is run by 517 tests, including 27 in the ba0 library and 65 in the DifferentialAlgebra package.

`void ba0_move_to_tail_table (struct ba0_table *T, struct ba0_table *U, ba0_int_p g)` [Function]
Perform a left rotation on the elements of U whose indices are greater than or equal to g, so that the element of U which was formerly at index g, occurs at the end of U. Result in T. Exception `BA0_ERRALG` is raised if g is negative or greater than the number of used elements of T. This function belongs to `ba0_table.c`. It is run by 139 tests, including 0 in the ba0 library and 30 in the DifferentialAlgebra package.

`void ba0_move_from_tail_table (struct ba0_table *T, struct ba0_table *U, ba0_int_p g)` [Function]
The inverse of the former operation. Performs a right rotation on the elements of U whose indices are greater than or equal to g, so that the element of U which was formerly at the end of U, occurs at index g. Result in T. Exception `BA0_ERRALG` is raised if g is negative or greater than the number of used elements of T. This function belongs to `ba0_table.c`. It is run by 148 tests, including 0 in the ba0 library and 30 in the DifferentialAlgebra package.

`void ba0_sort_table (struct ba0_table *dst, struct ba0_table *src)` [Function]
Sort the `tab` field of *src* by increasing addresses. Result in *dst*. This function is used by the bmi library. This function belongs to `ba0_table.c`. It is run by 18 tests, including 0 in the ba0 library and 18 in the DifferentialAlgebra package.

`void ba0_unique_table (struct ba0_table *T, struct ba0_table *U)` [Function]

Remove duplicated elements from U. The table U is supposed to be sorted. Result in T. This function is used by the `bmi` library. This function belongs to `ba0_table.c`. It is run by 18 tests, including 0 in the `ba0` library and 18 in the `DifferentialAlgebra` package.

4.10.2 Tables of machine integers

Description of type `ba0_tableof_int_p`. This type declaration belongs to `ba0_tableof_int_p.h`.

```
struct ba0_tableof_int_p
{
  ba0_int_p alloc;
  ba0_int_p size;
  ba0_int_p *tab;
};
```

`void ba0_init_tableof_int_p (struct ba0_tableof_int_p *table)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 546 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_tableof_int_p (struct ba0_tableof_int_p *table)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 491 tests, including 6 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`struct ba0_tableof_int_p * ba0_new_tableof_int_p (void)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 282 tests, including 2 in the `ba0` library and 41 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_tableof_int_p (struct ba0_tableof_int_p *table, enum ba0_garbage_code code)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 75 tests, including 1 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_realloc_tableof_int_p (struct ba0_tableof_int_p *table, ba0_int_p n)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 502 tests, including 14 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_re_malloc_tableof_int_p (struct ba0_tableof_int_p *table, ba0_int_p n)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 546 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_delete_tableof_int_p (struct ba0_tableof_int_p *table, ba0_int_p index)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 1 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_insert_tableof_int_p (struct ba0_tableof_int_p *table, ba0_int_p index, ba0_int_p value)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is not run by any test.

`bool ba0_member_tableof_int_p (ba0_int_p value, struct ba0_tableof_int_p *table)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 117 tests, including 0 in the `ba0` library and 19 in the `DifferentialAlgebra` package.

`bool ba0_member2_tableof_int_p (ba0_int_p value, struct ba0_tableof_int_p *table, ba0_int_p *index)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is not run by any test.

`void ba0_set_tableof_int_p (struct ba0_tableof_int_p *dst, struct ba0_tableof_int_p *src)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 419 tests, including 9 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_append_tableof_int_p_listof_int_p (struct ba0_tableof_int_p *table, struct ba0_listof_int_p *list)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is run by 62 tests, including 6 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_move_to_tail_tableof_int_p (struct ba0_tableof_int_p *dst, struct ba0_tableof_int_p *src, ba0_int_p index)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is not run by any test.

`void ba0_move_from_tail_tableof_int_p (struct ba0_tableof_int_p *dst, struct ba0_tableof_int_p *src, ba0_int_p index)` [Function]

This function belongs to `ba0_tableof_int_p.c`. It is not run by any test.

4.10.3 Tables of floating point numbers

Description of type `ba0_tableof_double`. This type declaration belongs to `ba0_tableof_double.h`.

```
struct ba0_tableof_double
{
  ba0_int_p alloc;
  ba0_int_p size;
  double *tab;
};
```

`void ba0_init_tableof_double (struct ba0_tableof_double *table)` [Function]

This function belongs to `ba0_tableof_double.c`. It is run by 501 tests, including 13 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_tableof_double (struct ba0_tableof_double *table)` [Function]

This function belongs to `ba0_tableof_double.c`. It is run by 228 tests, including 3 in the `ba0` library and 27 in the `DifferentialAlgebra` package.

`struct ba0_tableof_double * ba0_new_tableof_double (void)` [Function]
This function belongs to `ba0_tableof_double.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_tableof_double (struct ba0_tableof_double *table, enum ba0_garbage_code code)` [Function]
This function belongs to `ba0_tableof_double.c`. It is run by 76 tests, including 2 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_realloc_tableof_double (struct ba0_tableof_double *table, ba0_int_p n)` [Function]
This function belongs to `ba0_tableof_double.c`. It is run by 496 tests, including 11 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_re_malloc_tableof_double (struct ba0_tableof_double *table, ba0_int_p n)` [Function]
This function belongs to `ba0_tableof_double.c`. It is not run by any test.

`void ba0_delete_tableof_double (struct ba0_tableof_double *table, ba0_int_p index)` [Function]
This function belongs to `ba0_tableof_double.c`. It is not run by any test.

`void ba0_insert_tableof_double (struct ba0_tableof_double *table, ba0_int_p index, double value)` [Function]
This function belongs to `ba0_tableof_double.c`. It is not run by any test.

`bool ba0_member_tableof_double (double value, struct ba0_tableof_double *table)` [Function]
This function belongs to `ba0_tableof_double.c`. It is not run by any test.

`bool ba0_member2_tableof_double (double value, struct ba0_tableof_double *table, ba0_int_p *index)` [Function]
This function belongs to `ba0_tableof_double.c`. It is not run by any test.

`void ba0_set_tableof_double (struct ba0_tableof_double *dst, struct ba0_tableof_double *src)` [Function]
This function belongs to `ba0_tableof_double.c`. It is run by 495 tests, including 10 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_append_tableof_double_listof_double (struct ba0_tableof_double *table, struct ba0_listof_double *list)` [Function]
This function belongs to `ba0_tableof_double.c`. It is run by 15 tests, including 3 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_move_to_tail_tableof_double (struct ba0_tableof_double *dst, struct ba0_tableof_double *src, ba0_int_p index)` [Function]
This function belongs to `ba0_tableof_double.c`. It is not run by any test.

`void ba0_move_from_tail_tableof_double (struct ba0_tableof_double *dst, struct ba0_tableof_double *src, ba0_int_p index)` [Function]

This function belongs to `ba0_tableof_double.c`. It is not run by any test.

4.11 Matrices

4.11.1 Generic matrices

Description of type `ba0_typeof_matrix_memory_layout`. This data type permits to select the memory layout of matrices. This type declaration belongs to `ba0_matrix.h`.

```
enum ba0_typeof_matrix_memory_layout
{
  ba0_C_memory_layout,
  ba0_Fortran_memory_layout
};
```

Description of type `ba0_matrix`. This data type implements dense matrices. The fields `rowdim` and `coldim` provide the dimension of the matrix. If different from -1, the field `alloc` contains the number of allocated entries to `entry` and satisfies: `alloc` $\geq$ `rowdim` $\times$ `coldim`. It is equal to -1 if the matrix cannot be reallocated (see the `BA0_MATRIXOF_DOUBLE_VIEW` macro). The field `layout` indicates the memory layout used for the matrix and handled by the `BA0_MAT` macro. Default value: `ba0_Fortran_memory_layout`. The field `entry` points to the entries. If the format identifier for matrix elements is `%elt`, then the format identifier for the matrix is `%m[%elt]`. This type declaration belongs to `ba0_matrix.h`.

```
struct ba0_matrix
{
  ba0_int_p alloc;
  ba0_int_p rowdim;
  ba0_int_p coldim;
  enum ba0_typeof_matrix_memory_layout layout;
  void **entry;
};
```

`BA0_MAT (A, i, j) struct ba0_matrix A, ba0_int_p i, j` [Macro]

Denote the entry of M at row i and columns j. Indices start at zero.

`void ba0_init_matrix (struct ba0_matrix *M)` [Function]

Initialize M to the empty matrix (constructor). This function belongs to `ba0_matrix.c`. It is run by 35 tests, including 8 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`struct ba0_matrix * ba0_new_matrix (void)` [Function]

Allocate a new matrix, apply the constructor over it and return it. This function belongs to `ba0_matrix.c`. It is run by 19 tests, including 8 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_set_matrix (struct ba0_matrix *dst, struct ba0_matrix *src)` [Function]

Assign *src* to *dst*. This function belongs to `ba0_matrix.c`. It is run by 3 tests, including 3 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_realloc_matrix (struct ba0_matrix *M, ba0_int_p n, ba0_int_p m)` [Function]

If needed, reallocate M so that the matrix can contain $n \times m$ elements. The fields `rowdim` and `coldim` are updated. Existing elements are kept. This function belongs to `ba0_matrix.c`. It is run by 24 tests, including 8 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_realloc2_matrix (struct ba0_matrix *M, ba0_int_p n, ba0_int_p m, ba0_new_function *new_object)` [Function]

Variant of `ba0_realloc_matrix`. New elements are initialized with the value returned by *new_object*. This function belongs to `ba0_matrix.c`. It is run by 25 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_Bareiss_matrix (void *determinant, struct ba0_matrix *A, ba0_new_function *new_object, ba0_unary_operation *set_object_zero, ba0_unary_operation *set_object_one, ba0_unary_predicate *is_zero_object, ba0_binary_operation *set_object, ba0_binary_operation *neg_object, ba0_ternary_operation *sub_object, ba0_ternary_operation *mul_object, ba0_ternary_operation *exquo_object)` [Function]

Apply the Bareiss (fraction free Gaussian elimination) algorithm over matrix A. If nonzero, *determinant* is assigned the determinant of A (assuming A is square). All the following functions are supposed to apply to matrix elements. The functions *set_object_zero* and *set_object_one* are supposed to assign 0 or 1 to their argument. The function *is_zero_object* is supposed to return `true` if its argument is equal to 0 else `false`. The functions *set_object* and *neg_object* are supposed to assign their second argument and the opposite of their second argument to their first ones. The functions *sub_object*, *mul_object* and *exquo_object* are supposed to assign the difference, the product and the exact quotient of their second arguments with their third ones to their first ones. This function belongs to `ba0_matrix.c`. It is run by 6 tests, including 4 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.11.2 Matrices of machine integers

Description of type `ba0_matrixof_int_p`. This type declaration belongs to `ba0_matrixof_int_p.h`.

```
struct ba0_matrixof_int_p
{
  ba0_int_p alloc;
  ba0_int_p rowdim;
  ba0_int_p coldim;
  enum ba0_typeof_matrix_memory_layout layout;
  ba0_int_p *entry;
};
```

`void ba0_init_matrixof_int_p (struct ba0_matrixof_int_p *M)` [Function]

Initialize M to the empty matrix (constructor). This function belongs to `ba0_matrixof_int_p.c`. It is run by 2 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`struct ba0_matrixof_int_p * ba0_new_matrixof_int_p (void)` [Function]
Allocate a new matrix, apply the constructor over it and return it. This function belongs to `ba0_matrixof_int_p.c`. It is run by 2 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_set_matrixof_int_p (struct ba0_matrixof_int_p *dst, struct ba0_matrixof_int_p *src)` [Function]
Assign *src* to *dst*. This function belongs to `ba0_matrixof_int_p.c`. It is run by 2 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_realloc_matrixof_int_p (struct ba0_matrixof_int_p *M, ba0_int_p n, ba0_int_p m)` [Function]
If needed, reallocate M so that the matrix can contain $n \times m$ elements. The fields `rowdim` and `coldim` are updated. Existing elements are kept. This function belongs to `ba0_matrixof_int_p.c`. It is run by 2 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.11.3 Matrices of floating point numbers

Description of type `ba0_matrixof_double`. This type declaration belongs to `ba0_matrixof_double.h`.

```
struct ba0_matrixof_double
{
  ba0_int_p alloc;
  ba0_int_p rowdim;
  ba0_int_p coldim;
  enum ba0_typeof_matrix_memory_layout layout;
  double *entry;
};
```

`void ba0_init_matrixof_double (struct ba0_matrixof_double *M)` [Function]
Initialize M to the empty matrix (constructor). This function belongs to `ba0_matrixof_double.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`struct ba0_matrixof_double * ba0_new_matrixof_double (void)` [Function]
Allocate a new matrix, apply the constructor over it and return it. This function belongs to `ba0_matrixof_double.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_set_matrixof_double (struct ba0_matrixof_double *dst, struct ba0_matrixof_double *src)` [Function]
Assign *src* to *dst*. This function belongs to `ba0_matrixof_double.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_realloc_matrixof_double (struct ba0_matrixof_double *M, ba0_int_p n, ba0_int_p m)` [Function]
If needed, reallocate M so that the matrix can contain $n \times m$ elements. The fields `rowdim` and `coldim` are updated. Existing elements are kept. This function belongs to `ba0_matrixof_double.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.12 Values

4.12.1 Generic values

Description of type `ba0_value`. This data type permits to associate a value to a variable (variables are defined in the `bav` library).

It can be parsed and printed using formats of the form `%value(%something)`. The input output syntax for values is `var = value`. The equality sign can be customized (see `ba0_set_settings_value`). This type declaration belongs to `ba0_value.h`.

```
struct ba0_value
{
  struct bav_variable *var;
  void *pointer_data;
};
```

`void ba0_set_settings_value (char *equal_sign)` [Function]
Set to *equal_sign* the operator used for displaying values. This function belongs to `ba0_value.c`. It is run by 66 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_value (char **equal_sign)` [Function]
Assign to **equal_sign* the operator used to print values. This function belongs to `ba0_value.c`. It is not run by any test.

`void ba0_init_value (struct ba0_value *value)` [Function]
Initialize *value* to the pair (`BA0_NOT_A_VARIABLE`, 0). Constructor. This function belongs to `ba0_value.c`. It is run by 207 tests, including 0 in the `ba0` library and 28 in the `DifferentialAlgebra` package.

`struct ba0_value * ba0_new_value (void)` [Function]
Allocate a new value, initialize it and return the result. This function belongs to `ba0_value.c`. It is run by 207 tests, including 0 in the `ba0` library and 28 in the `DifferentialAlgebra` package.

4.12.2 Values of machine integers

Description of type `ba0_value_int_p`. This data type permits to associate a `ba0_int_p` value to a variable (variables are defined in the `bav` library).

It can be parsed and printed using formats of the form `%value(%d)`. The input output syntax for values is `var = value`. The equality sign can be customized (see `ba0_set_settings_value`). This type declaration belongs to `ba0_value_int_p.h`.

```
struct ba0_value_int_p
{
  struct bav_variable *var;
  ba0_int_p integer_data;
};
```

`void ba0_init_value_int_p (struct ba0_value_int_p *value)` [Function]
Initialize *value* to the pair (`BA0_NOT_A_VARIABLE`, 0). Constructor. This function belongs to `ba0_value_int_p.c`. It is run by 105 tests, including 0 in the `ba0` library and 19 in the `DifferentialAlgebra` package.

`struct ba0_value_int_p * ba0_new_value_int_p (void)` [Function]
Allocate a new value, initialize it and return the result. This function belongs to `ba0_value_int_p.c`. It is run by 105 tests, including 0 in the `ba0` library and 19 in the `DifferentialAlgebra` package.

4.12.3 Values of floating point numbers

Description of type `ba0_value_double`. This data type permits to associate a `double` value to a variable (variables are defined in the `bav` library).

It can be parsed and printed using formats of the form `%value(%le)`. The input output syntax for values is `var = value`. The equality sign can be customized (see `ba0_set_settings_value`). This type declaration belongs to `ba0_value_double.h`.

```
struct ba0_value_double
{
  struct bav_variable *var;
  double double_data;
};
```

`void ba0_init_value_double (struct ba0_value_double *value)` [Function]
Initialize *value* to the pair (`BA0_NOT_A_VARIABLE`, 0). Constructor. This function belongs to `ba0_value_double.c`. It is run by 13 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`struct ba0_value_double * ba0_new_value_double (void)` [Function]
Allocate a new value, initialize it and return the result. This function belongs to `ba0_value_double.c`. It is run by 13 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.13 Points

4.13.1 Generic points

Description of type `ba0_point`. This data type permits to associate values to many different variables. It actually is a duplicate of `struct ba0_table` so that many table functions may be applied to points. Many functions require the `tab` field to be sorted (see `ba0_sort_point`).

It can be parsed and printed using formats of the form `%point(%something)` which is more precise than `%t(%value(%something))` since parsed points are sorted and tested against ambiguity (exception `BA0_ERRAMB` is raised by the parser if the variables are not pairwise distinct).

This data type gets specialized as `struct bav_point_int_p` and `struct bav_point_interval_mpq` in the `bav` library and as `baz_point_ratfrac` in the `baz` library. This type declaration belongs to `ba0_point.h`.

```
struct ba0_point
{
  ba0_int_p alloc;
  ba0_int_p size;
  struct ba0_value **tab;
};
```

`void ba0_init_point (struct ba0_point *point)` [Function]
Initialize *point* to the empty point. This function belongs to `ba0_point.c`. It is run by 208 tests, including 0 in the `ba0` library and 28 in the `DifferentialAlgebra` package.

`void ba0_reset_point (struct ba0_point *point)` [Function]
reset *point* to the empty point. This function belongs to `ba0_point.c`. It is run by 63 tests, including 0 in the `ba0` library and 10 in the `DifferentialAlgebra` package.

`void ba0_realloc_point (struct ba0_point *point, ba0_int_p n)` [Function]
Reallocate *point* so that it may receive at least *n* entries. This function belongs to `ba0_point.c`. It is run by 196 tests, including 0 in the `ba0` library and 28 in the `DifferentialAlgebra` package.

`void ba0_set_point_list (struct ba0_point *point, struct ba0_list *L)` [Function]
The list *L* is supposed to be a list of `struct ba0_value`. Assign the elements of *L* to *point*. The elements of *L* (the values) are not duplicated. This function belongs to `ba0_point.c`. It is run by 33 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_set_point (struct ba0_point *dst, struct ba0_point *src)` [Function]
Store a copy of *src* to *dst*. The `struct ba0_value` *src* points to are duplicated but the `value` fields of the `struct ba0_value` are not. This function belongs to `ba0_point.c`. It is run by 124 tests, including 0 in the `ba0` library and 11 in the `DifferentialAlgebra` package.

`void ba0_delete_point (struct ba0_point *dst, struct ba0_point *src, ba0_int_p index)` [Function]
Assign to *dst* a copy of *src* without its entry number *index*. This function belongs to `ba0_point.c`. It is run by 66 tests, including 0 in the `ba0` library and 11 in the `DifferentialAlgebra` package.

`void ba0_sort_point (struct ba0_point *dst, struct ba0_point *src)` [Function]
Sort the `tab` field of *src* by increasing addresses of `var` fields. This ordering is the one expected by many functions. Result in *dst*. This function belongs to `ba0_point.c`. It is run by 208 tests, including 0 in the `ba0` library and 28 in the `DifferentialAlgebra` package.

`bool ba0_is_sorted_point (struct ba0_point *point)` [Function]
Return true if `ba0_bsearch_point` can be applied to *point*, false otherwise. This function belongs to `ba0_point.c`. It is not run by any test.

`struct ba0_value * ba0_assoc_point (struct bav_variable *var, struct ba0_point *point, ba0_int_p *index)` [Function]
Return a value whose `var` field is *var*, the zero pointer if no such value exists. If *index* is nonzero and a nonzero value is returned, its index is stored in **index*. This function belongs to `ba0_point.c`. It is not run by any test.

`struct ba0_value * ba0_bsearch_point (struct bav_variable *var, struct ba0_point *point, ba0_int_p *index)` [Function]

Only applies to sorted points. Return a value whose `var` field is *var*, the zero pointer if no such value exists. If *index* is nonzero and a nonzero value is returned, its index is stored in **index*. If *index* is nonzero the zero pointer is returned, **index* receives the index in `tab` which should receive *var*. This function belongs to `ba0_point.c`. It is run by 199 tests, including 0 in the `ba0` library and 28 in the `DifferentialAlgebra` package.

4.13.2 Points with machine integer values

Description of type `ba0_point_int_p`. This data type permits to associate values to many different variables. It actually is a duplicate of `struct ba0_table` so that many table functions may be applied to points. Many functions require the `tab` field to be sorted (see `ba0_sort_point_int_p`).

It can be parsed and printed using formats of the form `%point(%d)` which is more precise than `%t(%value(%d))` since parsed points are sorted and tested against ambiguity (exception `BA0_ERRAMB` is raised by the parser if the variables are not pairwise distinct). This type declaration belongs to `ba0_point_int_p.h`.

```
struct ba0_point_int_p
{
  ba0_int_p alloc;
  ba0_int_p size;
  struct ba0_value_int_p **tab;
};
```

`void ba0_init_point_int_p (struct ba0_point_int_p *point)` [Function]

Initialize *point* to the empty point. This function belongs to `ba0_point_int_p.c`. It is run by 124 tests, including 0 in the `ba0` library and 26 in the `DifferentialAlgebra` package.

`void ba0_realloc_point_int_p (struct ba0_point_int_p *point, ba0_int_p n)` [Function]

Reallocate *point* so that it may receive at least *n* entries. This function belongs to `ba0_point_int_p.c`. It is run by 122 tests, including 0 in the `ba0` library and 26 in the `DifferentialAlgebra` package.

`void ba0_set_point_int_p (struct ba0_point_int_p *dst, struct ba0_point_int_p *src)` [Function]

Store a copy of *src* to *dst*. The `struct ba0_value_int_p` *src* points to are duplicated but the `value` fields of the `struct ba0_value_int_p` are not. This function belongs to `ba0_point_int_p.c`. It is run by 1 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_delete_point_int_p (struct ba0_point_int_p *dst, struct ba0_point_int_p *src, ba0_int_p index)` [Function]

Assign to *dst* a copy of *src* without its entry number *index*. This function belongs to `ba0_point_int_p.c`. It is not run by any test.

`void ba0_sort_point_int_p (struct ba0_point_int_p *dst, struct ba0_point_int_p *src)` [Function]

Sort the `tab` field of *src* by increasing addresses of `var` fields. This ordering is the one expected by many functions. Result in *dst*. This function belongs to `ba0_point_int_p.c`. It is run by 122 tests, including 0 in the `ba0` library and 26 in the `DifferentialAlgebra` package.

`bool ba0_is_sorted_point_int_p (struct ba0_point_int_p *point)` [Function]

Return true if `ba0_bsearch_point_int_p` can be applied to *point*, false otherwise. This function belongs to `ba0_point_int_p.c`. It is not run by any test.

`struct ba0_value_int_p * ba0_assoc_point_int_p (struct bav_variable *var, struct ba0_point_int_p *point, ba0_int_p *index)` [Function]

Return a value whose `var` field is *var*, the zero pointer if no such value exists. If *index* is nonzero and a nonzero value is returned, its index is stored in **index*. This function belongs to `ba0_point_int_p.c`. It is not run by any test.

`struct ba0_value_int_p * ba0_bsearch_point_int_p (struct bav_variable *var, struct ba0_point_int_p *point, ba0_int_p *index)` [Function]

Only applies to sorted points. Return a value whose `var` field is *var*, the zero pointer if no such value exists. If *index* is nonzero and a nonzero value is returned, its index is stored in **index*. If *index* is nonzero the zero pointer is returned, **index* receives the index in `tab` which should receive *var*. This function belongs to `ba0_point_int_p.c`. It is run by 103 tests, including 0 in the `ba0` library and 19 in the `DifferentialAlgebra` package.

4.13.3 Points with floating point values

Description of type `ba0_point_double`. This data type permits to associate values to many different variables. It actually is a duplicate of `struct ba0_table` so that many table functions may be applied to points. Many functions require the `tab` field to be sorted (see `ba0_sort_point_double`).

It can be parsed and printed using formats of the form `%point(%le)` which is more precise than `%t(%value(%le))` since parsed points are sorted and tested against ambiguity (exception `BA0_ERRAMB` is raised by the parser if the variables are not pairwise distinct). This type declaration belongs to `ba0_point_double.h`.

```
struct ba0_point_double
{
  ba0_int_p alloc;
  ba0_int_p size;
  struct ba0_value_double **tab;
};
```

`void ba0_init_point_double (struct ba0_point_double *point)` [Function]

Initialize *point* to the empty point. This function belongs to `ba0_point_double.c`. It is run by 13 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_realloc_point_double (struct ba0_point_double *point, ba0_int_p n)` [Function]

Reallocate *point* so that it may receive at least *n* entries. This function belongs to `ba0_point_double.c`. It is not run by any test.

`void ba0_set_point_double (struct ba0_point_double *dst, struct ba0_point_double *src)` [Function]

Store a copy of *src* to *dst*. The `struct ba0_value_double` *src* points to are duplicated but the `value` fields of the `struct ba0_value_double` are not. This function belongs to `ba0_point_double.c`. It is not run by any test.

`void ba0_delete_point_double (struct ba0_point_double *dst, struct ba0_point_double *src, ba0_int_p index)` [Function]

Assign to *dst* a copy of *src* without its entry number *index*. This function belongs to `ba0_point_double.c`. It is not run by any test.

`void ba0_sort_point_double (struct ba0_point_double *dst, struct ba0_point_double *src)` [Function]

Sort the `tab` field of *src* by increasing addresses of `var` fields. This ordering is the one expected by many functions. Result in *dst*. This function belongs to `ba0_point_double.c`. It is not run by any test.

`bool ba0_is_sorted_point_double (struct ba0_point_double *point)` [Function]

Return true if `ba0_bsearch_point_double` can be applied to *point*, false otherwise. This function belongs to `ba0_point_double.c`. It is not run by any test.

`struct ba0_value_double * ba0_assoc_point_double (struct bav_variable *var, struct ba0_point_double *point, ba0_int_p *index)` [Function]

Return a value whose `var` field is *var*, the zero pointer if no such value exists. If *index* is nonzero and a nonzero value is returned, its index is stored in **index*. This function belongs to `ba0_point_double.c`. It is not run by any test.

`struct ba0_value_double * ba0_bsearch_point_double (struct bav_variable *var, struct ba0_point_double *point, ba0_int_p *index)` [Function]

Only applies to sorted points. Return a value whose `var` field is *var*, the zero pointer if no such value exists. If *index* is nonzero and a nonzero value is returned, its index is stored in **index*. If *index* is nonzero the zero pointer is returned, **index* receives the index in `tab` which should receive *var*. This function belongs to `ba0_point_double.c`. It is not run by any test.

4.14 Numbers

Numbers are machine integers, floating point numbers or some `GMP` objects.

4.14.1 Machine integers

The machine integers are implemented using the `ba0_int_p` data type.

`void ba0_set_tableof_tableof_int_p (struct ba0_tableof_tableof_int_p *dst, struct ba0_tableof_tableof_int_p *src)` [Function]

Assign *src* to *dst*. This function belongs to `ba0_int_p.c`. It is not run by any test.

`unsigned ba0_int_p ba0_sizeof_tableof_tableof_int_p (struct ba0_tableof_tableof_int_p *T, enum ba0_garbage_code code)` [Function]

Return the size needed to perform a copy of T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `ba0_int_p.c`. It is not run by any test.

`ba0_int_p ba0_log2_int_p (ba0_int_p size)` [Function]

Return the largest nonnegative integer e such that $2^e \leq size$. Return zero if *size* is not positive. This function belongs to `ba0_int_p.c`. It is run by 384 tests, including 0 in the `ba0` library and 63 in the `DifferentialAlgebra` package.

`void * ba0_scanf_int_p (void *z)` [Function]

General parsing functions for machine integers. It can be called using `ba0_scanf/%d`. This function belongs to `ba0_int_p.c`. It is run by 281 tests, including 8 in the `ba0` library and 3 in the `DifferentialAlgebra` package.

`void ba0_printf_int_p (void *z)` [Function]

General printing function for machine integers. It can be called using `ba0_printf/%d`. This function belongs to `ba0_int_p.c`. It is run by 314 tests, including 6 in the `ba0` library and 52 in the `DifferentialAlgebra` package.

`void * ba0_scanf_hexint_p (void *z)` [Function]

General parsing function for machine integers in hexadecimal notation. It can be called using `ba0_scanf/%x`. This function belongs to `ba0_int_p.c`. It is not run by any test.

`void ba0_printf_hexint_p (void *z)` [Function]

General printing function for machine integers in hexadecimal notation. It can be called using `ba0_printf/%x`. This function belongs to `ba0_int_p.c`. It is not run by any test.

4.14.2 Bools

The bools are implemented using the `ba0_bool` data type, which is an alias for `ba0_int_p`. The corresponding format is `%bool`.

`void * ba0_scanf_bool (void *z)` [Function]

General parsing functions for bools. It can be called using `ba0_scanf/%bool`. The bool can be entered as an `int`: a nonzero value is `true` while zero is `false`. It can also be entered as `true` or `false`. Exception `BA0_ERRBOOL` is raised in case of a syntax error. This function belongs to `ba0_bool.c`. It is run by 52 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_printf_bool (void *z)` [Function]
General printing function for machine integers. It can be called using `ba0_printf/%bool`. The bool is printed as `true` or `false`. This function belongs to `ba0_bool.c`. It is run by 46 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.14.3 Big integers

Big integers are implemented using the `GMP mpz_t` numbers, redefined as `ba0_mpz_t`.

`ba0__mpz_struct * ba0_new_mpz (void)` [Function]
Return a `ba0_mpz_t` allocated on the current stack. This function belongs to `ba0_gmp.c`. It is run by 341 tests, including 10 in the `ba0` library and 53 in the `DifferentialAlgebra` package.

`void ba0_set_tableof_mpz (struct ba0_tableof_mpz *dst, struct ba0_tableof_mpz *src)` [Function]
Assign *src* to *dst*. This function belongs to `ba0_gmp.c`. It is run by 38 tests, including 0 in the `ba0` library and 8 in the `DifferentialAlgebra` package.

`void ba0_set_tableof_tableof_mpz (struct ba0_tableof_tableof_mpz *dst, struct ba0_tableof_tableof_mpz *src)` [Function]
Assign *src* to *dst*. This function belongs to `ba0_gmp.c`. It is run by 38 tests, including 0 in the `ba0` library and 8 in the `DifferentialAlgebra` package.

`void ba0_record_gmp_memory_functions (void)` [Function]
Record the GMP memory allocation functions for further restoration. This function belongs to `ba0_gmp.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_restore_gmp_memory_functions (void)` [Function]
Restore the GMP memory allocation functions previously recorded. Calls the function pointed to by the settings variables `ba0_gmp_set_memory_functions`. This function belongs to `ba0_gmp.c`. It is run by 548 tests, including 54 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void * ba0_scanf_mpz (void *z)` [Function]
The general parser for `ba0_mpz_t`. It can be called through `ba0_scanf/%z`. The expected syntax is: a sequence of digits or `factorial (nonnegative integer)`, possibly starting with a minus sign. Exception `BA0_ERRINT` may be raised. This function belongs to `ba0_gmp.c`. It is run by 420 tests, including 22 in the `ba0` library and 49 in the `DifferentialAlgebra` package.

`void ba0_printf_mpz (void *z)` [Function]
The general printing function for `ba0_mpz_t`. It can be called through `ba0_printf/%z`. This function belongs to `ba0_gmp.c`. It is run by 386 tests, including 18 in the `ba0` library and 49 in the `DifferentialAlgebra` package.

4.14.4 Modular machine integers

The modulus is stored in the global variable `ba0_mint_hp_module`. The global boolean `ba0_mint_hp_module_is_prime` indicates if the modulus is a prime number or not. If not specified, all computations hold in the quotient ring defined by the modulus. The format identifier is `%im`.

`bool ba0_domain_mint_hp (void)` [Function]
Return `true` if the quotient ring is a domain (indeed a field). This function belongs to `ba0_mint_hp.c`. It is run by 142 tests, including 0 in the `ba0` library and 23 in the `DifferentialAlgebra` package.

`void ba0_mint_hp_module_set (ba0_mint_hp` *p*`, bool` *prime*`)` [Function]
Give values to the two global variables. This function belongs to `ba0_mint_hp.c`. It is run by 159 tests, including 1 in the `ba0` library and 28 in the `DifferentialAlgebra` package.

`enum ba0_wang_code ba0_wang_mint_hp (ba0_mpq_t` *rat*`,` [Function]
`ba0_mint_hp a, ba0_int_hp *`*ddz*`)`
Applie the Paul Shyh-Horng Wang algorithm to convert a modular number to a rational. Denote n the modulus. Assigns to *rat* the unique rational number p/q such that $|p|, |q| < \sqrt{n/2}$. Returns `ba0_rational_found` if the rational exists. Returns `ba0_zero_divisor` if a rational is found with a denominator not relatively prime to n. In this case, $*ddz$ receives $q \wedge n$ if *ddz* is not the zero pointer. Returns `ba0_rational_not_found` the rational does not exist. This function belongs to `ba0_mint_hp.c`. It is not run by any test.

4.14.5 Modular big numbers

The modular big numbers are implemented using the `ba0_mpz_t` numbers. For legibility, the data type `ba0_mpzm_t` is provided. The modulus is stored in the global variable `ba0_mpzm_module`. The global boolean `ba0_mpzm_module_is_prime` indicates if the modulus is prime or not.

The global variable `ba0_mpzm_module` (and some related ones) are stored in `ba0_quiet_stack` and are thus protected from deletion by the garbage collector. They are automatically reinitialized by `ba0_restart`.

`ba0__mpz_struct * ba0_new_mpzm (void)` [Function]
Return a `mpz_t` allocated on the current stack. This function belongs to `ba0_mpzm.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`bool ba0_domain_mpzm (void)` [Function]
Return `true` if the factor ring is a domain (a field). This function belongs to `ba0_mpzm.c`. It is run by 67 tests, including 0 in the `ba0` library and 6 in the `DifferentialAlgebra` package.

`void ba0_mpzm_module_set_ui (unsigned ba0_int_p n, bool prime)` [Function]

Assign the two parameters to the global variables. This function belongs to `ba0_mpzm.c`. It is run by 4 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_mpzm_module_set (ba0_mpz_t n, bool prime)` [Function]

Variant of the above function. This function belongs to `ba0_mpzm.c`. It is run by 113 tests, including 1 in the `ba0` library and 21 in the `DifferentialAlgebra` package.

`void ba0_mpzm_module_mul (ba0_mpz_t n)` [Function]

Multiplie the modulus by n. This function belongs to `ba0_mpzm.c`. It is not run by any test.

`void ba0_mpzm_module_pow_ui (ba0_mpz_t n, unsigned ba0_int_p d, bool prime)` [Function]

Assign n^d to the modulus. The boolean indicates if n is prime. This function belongs to `ba0_mpzm.c`. It is run by 2 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`enum ba0_wang_code ba0_wang_mpzm (ba0_mpq_t rat, ba0_mpz_t a, ba0_mpz_t ddz)` [Function]

Applie the Paul Shyh-Horng Wang algorithm to convert a modular number to a rational. Denote n the modulus. Assigns to *rat* the unique rational number p/q such that $|p|, |q| < \sqrt{n/2}$. Returns `ba0_rational_found` if the rational exists. Returns `ba0_zero_divisor` if a rational is found with a denominator not relatively prime to n. In this case, *ddz* receives $q \wedge n$ if *ddz* is not the zero pointer. Returns `ba0_rational_not_found` the rational does not exist. This function belongs to `ba0_mpzm.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void * ba0_scanf_mpzm (void *z)` [Function]

The general parser for modular `ba0_mpz_t`. It can be called through `ba0_scanf/%zm`. This function belongs to `ba0_mpzm.c`. It is not run by any test.

`void ba0_printf_mpzm (void *z)` [Function]

The general parser for modular `ba0_mpz_t`. It can be called through `ba0_printf/%zm`. This function belongs to `ba0_mpzm.c`. It is not run by any test.

4.14.6 Big rational numbers

They are implemented using the `GMP` `mpq_t` data type, redefined as `ba0_mpq_t`.

`ba0__mpq_struct * ba0_new_mpq (void)` [Function]

Return a `mpq_t` allocated on the current stack. This function belongs to `ba0_gmp.c`. It is run by 56 tests, including 6 in the `ba0` library and 9 in the `DifferentialAlgebra` package.

`void * ba0_scanf_mpq (void *z)` [Function]

The general parser for `ba0_mpq_t`. It can be called through `ba0_scanf/%q`. The expected syntax is: `%z` or `%z/%z` or a floating point number. Exceptions `BA0_ERRRAT`,

BA0_ERRDPZ may be raised. This function belongs to ba0_gmp.c. It is run by 67 tests, including 7 in the ba0 library and 12 in the DifferentialAlgebra package.

void ba0_printf_mpq (void *z) [Function]
The general printing function for ba0_mpq_t. It can be called through ba0_printf/%q. This function belongs to ba0_gmp.c. It is run by 25 tests, including 9 in the ba0 library and 2 in the DifferentialAlgebra package.

4.14.7 Floating point numbers

double * ba0_new_double (void) [Function]
Allocate a double in the current stack and return its address. This function belongs to ba0_double.c. It is run by 1 tests, including 1 in the ba0 library and 0 in the DifferentialAlgebra package.

void * ba0_scanf_double (void *z) [Function]
The general parsing function for doubles. It can be called via ba0_scanf/%le. Abnormal doubles such as inf and nan are allowed. This function belongs to ba0_double.c. It is run by 24 tests, including 4 in the ba0 library and 0 in the DifferentialAlgebra package.

void ba0_printf_double (void *z) [Function]
The general printing function for doubles. It can be called via ba0_printf/%le. This function belongs to ba0_double.c. It is run by 20 tests, including 4 in the ba0 library and 0 in the DifferentialAlgebra package.

int ba0_isnan (double x) [Function]
Return true if x is nan else false. This function belongs to ba0_double.c. It is run by 12 tests, including 1 in the ba0 library and 0 in the DifferentialAlgebra package.

int ba0_isinf (double x) [Function]
Return true if x is inf or -inf else false. This function belongs to ba0_double.c. It is run by 5 tests, including 3 in the ba0 library and 2 in the DifferentialAlgebra package.

double ba0_atof (char *s) [Function]
Convert the string s into a double. The string s may contain nan, inf or -inf. Errors are not detected. This function belongs to ba0_double.c. It is run by 144 tests, including 6 in the ba0 library and 14 in the DifferentialAlgebra package.

4.14.8 Intervals with big rational bounds

The data structure struct ba0_interval_mpq provides intervals with big rational bounds. The format identifier is %qi. The declarations are as follows:

Description of type ba0_typeof_interval. This type declaration belongs to ba0_interval_mpq.h.

```
enum ba0_typeof_interval
{
  ba0_closed_interval,
```

```
  ba0_open_interval,
  ba0_empty_interval,
  ba0_infinite_interval,
  ba0_left_infinite_interval,
  ba0_right_infinite_interval
};
```

Description of type `ba0_interval_mpq`. This type declaration belongs to `ba0_interval_mpq.h`.

```
struct ba0_interval_mpq
{
  ba0_mpq_t a;
  ba0_mpq_t b;
  enum ba0_typeof_interval type;
};
```

The fields `a` and `b` provide the lower and upper bounds of the intervals. The field `type` provides the type. The type `ba0_empty_interval` corresponds to the empty interval. The type `ba0_infinite_interval` corresponds to $(-\infty, +\infty)$. For these two types, the fields `a` and `b` are irrelevant. The type `ba0_left_infinite_interval` corresponds to $(-\infty, b)$. The type `ba0_right_infinite_interval` corresponds to $(a, +\infty)$. For these two types, the interval is open on its finite end. The type `ba0_closed_interval` corresponds to a single rational number. This number is stored in both bounds. The type `ba0_open_interval` corresponds to an interval (a, b) with both ends open.

`bool ba0_domain_interval_mpq (void)` [Function]

Return `true`. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_init_interval_mpq (struct ba0_interval_mpq *R)` [Function]

Initialize R to the empty interval (constructor). This function belongs to `ba0_interval_mpq.c`. It is run by 47 tests, including 2 in the `ba0` library and 9 in the `DifferentialAlgebra` package.

`struct ba0_interval_mpq * ba0_new_interval_mpq (void)` [Function]

Allocate an interval, initialize it and return it. This function belongs to `ba0_interval_mpq.c`. It is run by 47 tests, including 2 in the `ba0` library and 9 in the `DifferentialAlgebra` package.

`void ba0_set_interval_mpq_si (struct ba0_interval_mpq *I, ba0_int_p n)` [Function]

Assign the closed interval n to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_set_interval_mpq_ui (struct ba0_interval_mpq *I, unsigned ba0_int_p n)` [Function]

Assign n to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_set_interval_mpq_double (struct ba0_interval_mpq *I, double d)` [Function]

Assign d to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_set_interval_mpq_mpq (struct ba0_interval_mpq *I, ba0_mpq_t a, ba0_mpq_t b)` [Function]

Assign the interval (a, b) to I. The interval is closed if the two bounds are equal, else it is open. This function belongs to `ba0_interval_mpq.c`. It is run by 7 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_set_interval_mpq_type_mpq (struct ba0_interval_mpq *I, enum ba0_typeof_interval type, ba0_mpq_t a, ba0_mpq_t b)` [Function]

Assign the interval (a, b), of type *type*, to I. Depending on *type*, some of the bounds may be irrelevant. If the interval is closed, then the two bounds must be equal. If it is open, then the lower bound must be strictly less than the upper one. This function belongs to `ba0_interval_mpq.c`. It is run by 22 tests, including 2 in the `ba0` library and 1 in the `DifferentialAlgebra` package.

`void ba0_set_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *J)` [Function]

Assign J to I. This function belongs to `ba0_interval_mpq.c`. It is run by 22 tests, including 2 in the `ba0` library and 1 in the `DifferentialAlgebra` package.

`bool ba0_is_empty_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I is empty, else `false`. This function belongs to `ba0_interval_mpq.c`. It is run by 9 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`bool ba0_is_unbounded_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if, at least, one of the bounds I is infinite. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`bool ba0_is_closed_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I is closed. This function belongs to `ba0_interval_mpq.c`. It is run by 16 tests, including 1 in the `ba0` library and 1 in the `DifferentialAlgebra` package.

`bool ba0_is_open_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I is open. This function belongs to `ba0_interval_mpq.c`. It is run by 5 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`bool ba0_is_zero_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I is the closed interval 0. This function belongs to `ba0_interval_mpq.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`bool ba0_is_one_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I is the closed interval 1. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`bool ba0_are_equal_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *J)` [Function]

Return `true` if I and J are equal. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`bool ba0_contains_zero_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I contains 0. This function belongs to `ba0_interval_mpq.c`. It is run by 2 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`bool ba0_is_positive_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I contains positive numbers only. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`bool ba0_is_nonnegative_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I contains nonnegative numbers only. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`bool ba0_is_negative_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I contains negative numbers only. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`bool ba0_is_nonpositive_interval_mpq (struct ba0_interval_mpq *I)` [Function]

Return `true` if I contains nonpositive numbers only. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`bool ba0_is_less_interval_mpq (struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y)` [Function]

Return `true` if all the elements of X are less than all the elements of Y. This function belongs to `ba0_interval_mpq.c`. It is run by 7 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`bool ba0_are_disjoint_interval_mpq (struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y)` [Function]

Return `true` if the intersection of X and Y is empty. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_intersect_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y)` [Function]

Assign the intersection of X and Y to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`bool ba0_member_interval_mpq (ba0_mpq_t q, struct ba0_interval_mpq *X)` [Function]

Return `true` if q belongs to X. This function belongs to `ba0_interval_mpq.c`. It is run by 8 tests, including 0 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`bool ba0_is_subset_interval_mpq (struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y)` [Function]

Return `true` if X is a subset of Y. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_element_interval_mpq (ba0_mpq_t q, struct ba0_interval_mpq *I, unsigned ba0_int_p k)` [Function]

Assign to q some element (the kth) of I. If I is closed then k must be 0. If I contains infinitely many elements, then, for each value of k, a different element of I is assigned to q. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_middle_interval_mpq (ba0_mpq_t q, struct ba0_interval_mpq *I)` [Function]

Assign to q the middle of I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`double ba0_middle_interval_mpq_double (struct ba0_interval_mpq *I)` [Function]

Return the middle of I, as a `double`. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_width_interval_mpq (ba0_mpq_t q, struct ba0_interval_mpq *I)` [Function]

Assign to q the width of I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`double ba0_width_interval_mpq_double (struct ba0_interval_mpq *I)` [Function]

Return the width of I, as a `double`. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_abs_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X)` [Function]

Assign $|X|$ to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_neg_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X)` [Function]

Assign $-X$ to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_add_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y)` [Function]

Assign $X + Y$ to I. This function belongs to `ba0_interval_mpq.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_add_interval_mpq_mpq (struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y, ba0_mpq_t k)` [Function]

Assign $Y + k$ to X. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_sub_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y)` [Function]

Assign $X - Y$ to I. This function belongs to `ba0_interval_mpq.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_sub_interval_mpq_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, ba0_mpq_t k)` [Function]

Assign $X - k$ to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_sub_mpq_interval_mpq (struct ba0_interval_mpq *I, ba0_mpq_t k, struct ba0_interval_mpq *Y)` [Function]

Assign $q - Y$ to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_mul_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y)` [Function]

Assign $X \times Y$ to I. This function belongs to `ba0_interval_mpq.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_mul_interval_mpq_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, ba0_mpq_t k)` [Function]

Assign $X \times k$ to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_mul_interval_mpq_si (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, ba0_int_p n)` [Function]

Assign $X \times n$ to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_mul_interval_mpq_ui (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, unsigned ba0_int_p n)` [Function]

Assign $X \times n$ to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_pow_interval_mpq (struct ba0_interval_mpq *I, struct ba0_interval_mpq *X, ba0_int_p n)` [Function]

Assign X^n to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_pow_interval_mpq_mpq (struct ba0_interval_mpq *I, ba0_mpq_t q, ba0_int_p n)` [Function]

Assign q^n to I. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_div_interval_mpq (struct ba0_tableof_interval_mpq *F, struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y)` [Function]

Assign X/Y to F. The result is either a single interval or a union of two disjoint intervals (when Y contains 0). These resulting intervals are stored in F. This function belongs to `ba0_interval_mpq.c`. It is run by 1 tests, including 1 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_div_mpq_interval_mpq (struct ba0_tableof_interval_mpq *F, ba0_mpq_t k, struct ba0_interval_mpq *Y)` [Function]

Assign k/Y to F. The result is either a single interval or a union of two disjoint intervals (when Y contains 0). These resulting intervals are stored in F. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void ba0_div_interval_mpq_mpq (struct ba0_interval_mpq *X, struct ba0_interval_mpq *Y, ba0_mpq_t k)` [Function]

Assign Y/k to X. This function belongs to `ba0_interval_mpq.c`. It is not run by any test.

`void * ba0_scanf_interval_mpq (void *z)` [Function]

Parsing function for intervals. Allowed intervals are either `empty`, a rational number or a pair of rational numbers $[a, b]$ with $a \leq b$. In this last case, if $a = b$ the interval is closed else it is open. This function is called by `ba0_scanf/%qi`. This function belongs to `ba0_interval_mpq.c`. It is run by 7 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void ba0_printf_interval_mpq (void *z)` [Function]

The general printing function for intervals. It is called by `ba0_printf/%qi`. This function belongs to `ba0_interval_mpq.c`. It is run by 7 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

4.15 Strings

4.15.1 Plain strings

The *string* type is the plain `char*` type. The following functions are provided for convenience.

`char * ba0_not_a_string (void)` [Function]

Return `(char*)0`. This function belongs to `ba0_string.c`. It is run by 487 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`char * ba0_new_string (void)` [Function]

Return the empty string allocated in the current stack (not the `(char*)0` pointer, which is not a string). This function belongs to `ba0_string.c`. It is not run by any test.

`char * ba0_strdup (char *s)` [Function]

Return a copy of s in an area dynamically allocated in the current stack. This function belongs to `ba0_string.c`. It is run by 496 tests, including 9 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`char * ba0_strcat (struct ba0_tableof_string *T)` [Function]

Concatenate all the strings of T in a single, allocated, string and return it. This function belongs to `ba0_string.c`. It is run by 35 tests, including 6 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`int ba0_strcasecmp (char *s1, char *s2)` [Function]
See the documentation of `strcasecmp` which is not always available, depending on compiler options. This function belongs to `ba0_string.c`. It is run by 498 tests, including 4 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`int ba0_strncasecmp (char *s1, char *s2, size_t n)` [Function]
See the documentation of `strncasecmp` which is not always available, depending on compiler options. This function belongs to `ba0_string.c`. It is run by 144 tests, including 6 in the `ba0` library and 14 in the `DifferentialAlgebra` package.

`void * ba0_scanf_string (void *z)` [Function]
General parsing function for strings. It can be called by `ba0_scanf/%s`. It actually returns a copy (allocated by `ba0_alloc`) of the current token (see the lexical analyzer). This function belongs to `ba0_string.c`. It is run by 507 tests, including 20 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_printf_string (void *str)` [Function]
General printing function for strings. It can be called using `ba0_printf/%s`. This function belongs to `ba0_string.c`. It is run by 301 tests, including 21 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_tableof_string (struct ba0_tableof_string *T, enum ba0_garbage_code code)` [Function]
Return the size needed to perform a copy of T, including the strings themselves. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `ba0_string.c`. It is run by 74 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_tableof_string (struct ba0_tableof_string *dst, struct ba0_tableof_string *src)` [Function]
Assigns *src* to *dst*. Strings are duplicated. This function belongs to `ba0_string.c`. It is run by 165 tests, including 2 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_member2_tableof_string (char *string, struct ba0_tableof_string *T, ba0_int_p *index)` [Function]
Return `true` if *string* belongs to T, else `false`. Comparisons are performed using `strcmp`. If *index* is nonzero and *string* is found then the index of *string* in T is assigned to **index*. This function belongs to `ba0_string.c`. It is not run by any test.

4.15.2 Indexed strings

Description of type `ba0_indexed_string`. This data type permits to represent *indexed strings* which are strings endowed with indexed string indices. It is used for parsing symbols and variables. Here are a few examples of indexed strings accepted by parsers: `u`, `u[1]`, `u[[1],-4,[[3]]]`, `u[x[e]]`.

```
INDEXED ::= string INDICES ... INDICES              (*)
```

```
::= signed integer INDICES ... INDICES
::= INDICES ... INDICES

INDICES ::= (INDEXED, ..., INDEXED)
::= [INDEXED, ..., INDEXED]

(*) At top level, this form is the only one accepted.
```

This type declaration belongs to `ba0_indexed_string.h`.

```
struct ba0_indexed_string
{
// the string or the signed integer as a string or (char *)0
  char *string;
// the table of the indexed string indices
  struct ba0_tableof_indexed_string_indices Tindic;
};
```

Description of type `ba0_tableof_indexed_string_indices`. This type declaration belongs to `ba0_indexed_string.h`.

```
struct ba0_tableof_indexed_string_indices
{
  ba0_int_p alloc;
  ba0_int_p size;
  struct ba0_indexed_string_indices **tab;
};
```

Description of type `ba0_indexed_string_indices`. This data type is a subtype of `struct ba0_indexed_string`. It permits to store *indexed string indices*. This type declaration belongs to `ba0_indexed_string.h`.

```
struct ba0_indexed_string_indices
{
// the opening and closing parenthesis
  char po, pf;
  struct ba0_tableof_indexed_string Tindex;
};
```

`void ba0_init_indexed_string (struct ba0_indexed_string *indexed)` [Function]

Initialize *indexed* to the empty indexed string. This function belongs to `ba0_indexed_string.c`. It is run by 502 tests, including 15 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_indexed_string (struct ba0_indexed_string *indexed)` [Function]

Reset *indexed* to the empty indexed string. This function belongs to `ba0_indexed_string.c`. It is run by 494 tests, including 8 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`struct ba0_indexed_string * ba0_new_indexed_string (void)` [Function]

Allocate a new indexed string, initialize it and return it. This function belongs to `ba0_indexed_string.c`. It is run by 502 tests, including 15 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_indexed_string (struct ba0_indexed_string *dst, struct ba0_indexed_string *src)` [Function]

Assign *src* to *dst*. This function belongs to `ba0_indexed_string.c`. It is not run by any test.

`bool ba0_has_empty_trailing_indices_indexed_string (struct ba0_indexed_string *indexed, char avoid)` [Function]

Return `true` if *indexed* has an empty last indexed string indices the opening parenthesis of which is different from *avoid*. This function belongs to `ba0_indexed_string.c`. It is not run by any test.

`bool ba0_has_trailing_indices_indexed_string (struct ba0_indexed_string *indexed, bool (*predicate) (char *))` [Function]

Return `true` if *indexed* has at least one indexed string indices and each element of the last sequence of indices satisfies *predicate*. This function belongs to `ba0_indexed_string.c`. It is run by 488 tests, including 2 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_has_numeric_trailing_indices_indexed_string (struct ba0_indexed_string *indexed, struct ba0_tableof_int_p *indices)` [Function]

Return `true` if the trailing indexed string indices of *indexed* exists, is nonempty and involves signed integer numbers only. If so and *indices* is nonzero then *indices* is assigned the corresponding table of integers. This function belongs to `ba0_indexed_string.c`. It is run by 140 tests, including 2 in the `ba0` library and 15 in the `DifferentialAlgebra` package.

`void ba0_printf_indexed_string (void *z)` [Function]

General printing function for indexed. It is called by `ba0_printf/%indexed_string`. This function belongs to `ba0_indexed_string.c`. It is run by 502 tests, including 15 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`char * ba0_indexed_string_to_string (struct ba0_indexed_string *indexed)` [Function]

Print *indexed* into a string (allocated by `ba0_alloc`) and return this string. This function belongs to `ba0_indexed_string.c`. It is run by 492 tests, including 6 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`char * ba0_stripped_indexed_string_to_string (struct ba0_indexed_string *indexed)` [Function]

Apply `ba0_indexed_string_to_string` to the indexed string obtained by removing the trailing indexed string indices from *indexed* and return the result. This function belongs to `ba0_indexed_string.c`. It is run by 137 tests, including 2 in the `ba0` library and 14 in the `DifferentialAlgebra` package.

`void * ba0_scanf_indexed_string_as_a_string (void *z)` [Function]

Parsing function which can be called by `ba0_scanf/%six`. Call `ba0_scanf_indexed_string` but return the result as a plain string. Store it in $*z$ if this parameter is nonzero. This function belongs to `ba0_indexed_string.c`. It is run by 498 tests, including 11 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`struct ba0_indexed_string *` [Function]
`ba0_scanf_indexed_string_with_counter (struct ba0_indexed_string *indexed, ba0_int_p *counter)`

Variant of `ba0_scanf_indexed_string` which assigns to **counter* the argument for `ba0_unget_token_analex` in order to put back on the lexical analyzer fifo all the tokens which led to the last indexed string indices of the returned indexed string. The content of **counter* is meaningless if the returned indexed string does not involve any indexed string indices. This function belongs to `ba0_indexed_string.c`. It is run by 291 tests, including 2 in the `ba0` library and 55 in the `DifferentialAlgebra` package.

`void * ba0_scanf_indexed_string (void *z)` [Function]

The general parsing function for indexed. It is called by `ba0_scanf/%indexed_string`. Return the result as a `struct ba0_indexed_string *`. Store it in **z* if this parameter is nonzero. This function belongs to `ba0_indexed_string.c`. It is run by 502 tests, including 15 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.15.3 Range indexed groups

Description of type `ba0_range_indexed_group`. This data structure permits to describe a group of *range indexed strings*, which are strings indexed by integer numbers running over ranges. These strings (without their indices) are called the *radicals* of the range indexed strings. The tables `lhs` and `rhs` have the same size, which give the number of *range indices*. Though stored in doubles, the left-hand and right-hand sides of the ranges are either signed integers or `inf` or `-inf`. The left-hand side of a range may be greater than, equal to or lower than the right-hand side. Here are a few examples:

```
1. y
2. (x)[17:-3]
3. (y,z)[0:inf,inf:-1]
```

The radicals of the range indexed strings are `y`, `x`, `y` and `z`. In Example 1, the range indexed group involves a single range indexed string which has no range indices. The range indexed group is said to describe a *plain string*. In Example 2, the range indexed group involves a single range indexed string which has a single range index. In Example 3, the range indexed group involves two range indexed strings; each of them admits two range indices.

The aim of this data structure is to describe possibly infinite ordered sets of variables. The range indexed group of Example 2 describes the ordered set: `x[17] > x[16] > ... > x[-2]` (the least index of the range is assumed to be excluded, as in Python, but this behaviour can be customized). The variable `x[16]` is said to *fit* the range indexed string `(x)[17:-3]`. The variables `x[18]` or `z[0]` do not. In Example 3, the set of variables which fit the range indexed group is made of `y` and `z`, indexed by two nonnegative integers. This type declaration belongs to `ba0_range_indexed_group.h`.

```
struct ba0_range_indexed_group
{
// The left-hand sides of the ranges
  struct ba0_tableof_double lhs;
// The right-hand sides of the ranges
```

```
  struct ba0_tableof_double rhs;
// The table of the radicals of the range indexed strings
  struct ba0_tableof_string strs;
};
```

`void ba0_set_settings_range_indexed_group (char *oper, char *infinity, bool rhs_included, bool protect_from_evaluation)` [Function]

Set the range operator and the string for infinity used for printing range indexed groups to *oper* and *infinity*. Set the bool which determines whether the right-hand side of range indices should be included or excluded in ranges to *rhs_included*. Set the bool which indicates whether range indexed groups which are not plain strings should be quoted when printed. All these arguments may be zero. In such cases, default values are restored. This function belongs to `ba0_range_indexed_group.c`. It is run by 67 tests, including 2 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_get_settings_range_indexed_group (char **oper, char **infinity, bool *rhs_included, bool *protect_from_evaluation)` [Function]

Assign to **oper* and **infinity* the strings used to print the range operator and the infinity string when printing. Assign to **rhs_included* the bool which determines whether the right-hand side of range indices should be included or not in the ranges. Assign to **protect_from_evaluation* the bool which indicates if the range indexed groups which are not plain strings should be quoted when printed. All arguments are allowed to be zero. This function belongs to `ba0_range_indexed_group.c`. It is run by 2 tests, including 0 in the `ba0` library and 2 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_range_indexed_group (struct ba0_range_indexed_group *G, enum ba0_garbage_code code, bool strings_not_copied)` [Function]

Return the size needed to perform a copy of *G*. If *code* is `ba0_embedded` then *G* is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* is `true` then the range indexed strings are supposed not to be copied. This function belongs to `ba0_range_indexed_group.c`. It is run by 76 tests, including 2 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_range_indexed_group (struct ba0_range_indexed_group *dst, struct ba0_range_indexed_group *src)` [Function]

Assign (a full copy of) *src* to *dst*. This function belongs to `ba0_range_indexed_group.c`. It is run by 494 tests, including 7 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_range_indexed_group_with_tableof_string (struct ba0_range_indexed_group *dst, struct ba0_range_indexed_group *src, struct ba0_dictionary_string *D, struct ba0_tableof_string *T)` [Function]

Assign *src* to *dst* without any duplication of the radicals of the range indexed strings. Indeed, each radical in *src* has a copy in *T* whose index is supposed to be registered

in *D*. Instead of duplicating strings, the copies are used. Exception `BA0_ERRALG` is raised if some string is not found in *D*. This function belongs to `ba0_range_indexed_group.c`. It is run by 489 tests, including 2 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_tableof_range_indexed_group` [Function]
`(struct ba0_tableof_range_indexed_group *T, enum ba0_garbage_code code, bool strings_not_copied)`

Return the size needed to perform a copy of *T*. If *code* is `ba0_embedded` then *T* is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* is `true` then the radicals of the range indexed strings of *T* are supposed not to be copied. This function belongs to `ba0_range_indexed_group.c`. It is run by 76 tests, including 2 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_tableof_string_tableof_range_indexed_group` [Function]
`(struct ba0_tableof_string *T, struct ba0_tableof_range_indexed_group *rigs)`

Store in *T* all the range indexed strings occurring in *rigs*. The strings are not duplicated. This function belongs to `ba0_range_indexed_group.c`. It is run by 2 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void` [Function]
`ba0_set_tableof_range_indexed_group_with_tableof_string`
`(struct ba0_tableof_range_indexed_group *dst, struct ba0_tableof_range_indexed_group *src, struct ba0_dictionary_string *D, struct ba0_tableof_string *T)`

Assign *src* to *dst* without any duplication of the radicals of the range indexed strings. Indeed, each radical in *src* has a copy in *T* whose index is supposed to be registered in *D*. Instead of duplicating strings, the copies are used. Exception `BA0_ERRALG` is raised if some string is not found in *D*. This function is called by `bav_R_set_differential_ring` and `bav_R_new_ordering`. This function belongs to `ba0_range_indexed_group.c`. It is run by 489 tests, including 2 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_init_range_indexed_group (struct ba0_range_indexed_group *G)` [Function]

Initialize *G* to the empty range indexed group. This function belongs to `ba0_range_indexed_group.c`. It is run by 500 tests, including 10 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_range_indexed_group (struct ba0_range_indexed_group *G)` [Function]

Reset *G* to the empty range indexed group. This function belongs to `ba0_range_indexed_group.c`. It is not run by any test.

`struct ba0_range_indexed_group *` [Function]
`ba0_new_range_indexed_group (void)`

Allocate a new range indexed group, initialize it and return it. This function belongs to `ba0_range_indexed_group.c`. It is run by 497 tests, including 10 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_range_indexed_group (struct ba0_range_indexed_group *dst, struct ba0_range_indexed_group *src)` [Function]

Assign (a full copy of) *src* to *dst*. This function belongs to `ba0_range_indexed_group.c`. It is run by 494 tests, including 7 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_range_indexed_group_string (struct ba0_range_indexed_group *G, char *string)` [Function]

Assign *string* to G as a plain string. The string is not duplicated. This function belongs to `ba0_range_indexed_group.c`. It is run by 225 tests, including 0 in the `ba0` library and 27 in the `DifferentialAlgebra` package.

`bool ba0_is_plain_string_range_indexed_group (struct ba0_range_indexed_group *G, char **string)` [Function]

Return `true` if G is a plain string, else `false`. In any case, if *string* is nonzero then **string* is assigned the first range indexed string of G. This function belongs to `ba0_range_indexed_group.c`. It is run by 492 tests, including 5 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`bool ba0_compatible_indices_range_indexed_group (struct ba0_range_indexed_group *G, struct ba0_range_indexed_group *H)` [Function]

This function belongs to `ba0_range_indexed_group.c`. It is not run by any test.

`bool ba0_fit_range_indexed_group (struct ba0_range_indexed_group *G, char *radical, struct ba0_tableof_int_p *indices, ba0_int_p *index)` [Function]

Return `true` if *radical*[*indices*] fits G else `false`. The string *radical* must be equal to some element of the field `strs` of G. Each i in *indices* must belong to the corresponding range of G. In the positive case, if *index* is nonzero, then **index* is assigned the index of *radical* in the field `strs` of G. This function belongs to `ba0_range_indexed_group.c`. It is run by 2 tests, including 2 in the `ba0` library and 0 in the `DifferentialAlgebra` package.

`void * ba0_scanf_range_indexed_group (void *z)` [Function]

General parsing function for range indexed groups. It can be called using `ba0_scanf/%range_indexed_group`. The grammar is:

```
RIG ::= <radical>
RIG ::= "(" <non-empty sequence of radicals ")"
"[" non-empty sequence of RANGE "]"
RANGE ::= <[-]integer | [-]infinity> <range operator>
<[-]integer | [-]infinity>
```

The accepted range operator may be customized but `":"` and `".."` are allowed. The accepted string for infinity may be customized but `"oo"`, `"inf"`, `"infty"` and `"infinity"` are allowed. This function belongs to `ba0_range_indexed_group.c`. It is run by 497 tests, including 10 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_printf_range_indexed_group (void *z)` [Function]

General printing function for range indexed groups. It can be called using `ba0_printf/%range_indexed_group`. The strings used for the range operator and the infinity can be customized. This function belongs to `ba0_range_indexed_group.c`. It is run by 34 tests, including 5 in the `ba0` library and 8 in the `DifferentialAlgebra` package.

4.16 Dictionaries

4.16.1 Dictionaries with generic pointer keys

Description of type `ba0_dictionary`. This data structure implements generic dictionaries which map `void *` pointers or `ba0_int_p` to other objects. In this implementation, the values associated to the keys are integers, which are supposed to be indices in some table of objects of unspecified type. It is implemented using the double hash strategy. The entries of the field `area` contain either -1 or the index, in some unspecified table, of some object. The size of `area` is a power of two given by `log2_size`.

For computing hash values, keys are first shifted `shift` bits to the right. For a dictionary of integers `shift` should be zero. For a dictionary of pointers to objects of type T, `shift` should be the highest exponent e such that 2^e is less than or equal to the size in bytes of the objects to type T. This type declaration belongs to `ba0_dictionary.h`.

```
struct ba0_dictionary
{
// each entry contains either -1 or an index in some table of variables
  struct ba0_tableof_int_p area;
// the shift applied on keys before hashing them
  ba0_int_p shift;
// the base-2 logarithm of the size/alloc field of area if area is nonempty
  ba0_int_p log2_size;
// the number of used entries in area
  ba0_int_p used_entries;
};
```

`void ba0_init_dictionary (struct ba0_dictionary *D, ba0_int_p sizelt, ba0_int_p log2_size)` [Function]

Initialize D to the empty dictionary. The argument *sizelt* contains the size in bytes of the elements the dictionary keys point to. The argument *log2_size* provides the initial size of the field `area` of D. It should be at least equal to 8. This function belongs to `ba0_dictionary.c`. It is run by 500 tests, including 6 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_dictionary (struct ba0_dictionary *D)` [Function]

Reset D to the empty dictionary. This function belongs to `ba0_dictionary.c`. It is run by 490 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_dictionary (struct ba0_dictionary *D, enum ba0_garbage_code code)` [Function]

Return the size needed to perform a copy of D. If *code* is `ba0_embedded` then D is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `ba0_dictionary.c`. It is run by 75 tests, including 1 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_dictionary (struct ba0_dictionary *dst, struct ba0_dictionary *src)` [Function]

Assign *src* to *dst*. This function belongs to `ba0_dictionary.c`. It is run by 334 tests, including 3 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`ba0_int_p ba0_get_dictionary (struct ba0_dictionary *D, struct ba0_table *T, void *key)` [Function]

Return the index in T of the element with identifier *key*. The element must have been recorded in D. Return `BA0_NOT_AN_INDEX` if not found. This function belongs to `ba0_dictionary.c`. It is run by 492 tests, including 5 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_add_dictionary (struct ba0_dictionary *D, struct ba0_table *T, void *key, ba0_int_p z)` [Function]

Add to D an entry for the zth element of T which has key *key*. The element may actually not yet be present in T. Exception `BA0_ERRKEY` is raised if the key is already present in D. This function belongs to `ba0_dictionary.c`. It is run by 493 tests, including 6 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.16.2 Dictionaries with machine integer keys

Description of type `ba0_dictionary_int_p`. This data structure implements dictionaries mapping `ba0_int_p` keys to indices in a `ba0_tableof_int_p`. It is implemented using the double hash strategy. The entries of the field `area` contain either -1 or an index in the associated table. The size of `area` is a power of two given by `log2_size`. This type declaration belongs to `ba0_dictionary_int_p.h`.

```
struct ba0_dictionary_int_p
{
// each entry contains either -1 or an index in a tableof_int_p
  struct ba0_tableof_int_p area;
// the base-2 logarithm of the size/alloc field of area if area is nonempty
  ba0_int_p log2_size;
// the number of used entries in area
  ba0_int_p used_entries;
};
```

`void ba0_init_dictionary_int_p (struct ba0_dictionary_int_p *D, ba0_int_p log2_size)` [Function]

Initialize D to the empty dictionary. The argument *log2_size* provides the initial size of the field `area` of D. It should be between 1 and 20. This function belongs to `ba0_dictionary_int_p.c`. It is run by 99 tests, including 0 in the `ba0` library and 10 in the `DifferentialAlgebra` package.

`void ba0_reset_dictionary_int_p (struct ba0_dictionary_int_p *D)` [Function]

Reset D to the empty dictionary. This function belongs to `ba0_dictionary_int_p.c`. It is run by 99 tests, including 0 in the `ba0` library and 10 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_dictionary_int_p (struct ba0_dictionary_int_p *D, enum ba0_garbage_code code)` [Function]

Return the size needed to perform a copy of D. If *code* is `ba0_embedded`, then D is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `ba0_dictionary_int_p.c`. It is not run by any test.

`void ba0_set_dictionary_int_p (struct ba0_dictionary_int_p *dst, struct ba0_dictionary_int_p *src)` [Function]

Assign *src* to *dst*. This function belongs to `ba0_dictionary_int_p.c`. It is not run by any test.

`ba0_int_p ba0_get_dictionary_int_p (struct ba0_dictionary_int_p *D, struct ba0_tableof_int_p *T, ba0_int_p key)` [Function]

Return the index in T of the element with key *key*. The element must have been recorded in D. Return `BA0_NOT_AN_INDEX` if not found. This function belongs to `ba0_dictionary_int_p.c`. It is run by 99 tests, including 0 in the `ba0` library and 10 in the `DifferentialAlgebra` package.

`void ba0_add_dictionary_int_p (struct ba0_dictionary_int_p *D, struct ba0_tableof_int_p *T, ba0_int_p key, ba0_int_p z)` [Function]

Add to D an entry for the zth element of T, whose key is *key*. The element may not yet be present in T. Exception `BA0_ERRKEY` is raised if the key is already present in D. This function belongs to `ba0_dictionary_int_p.c`. It is run by 99 tests, including 0 in the `ba0` library and 10 in the `DifferentialAlgebra` package.

4.16.3 Dictionaries with string keys

Description of type `ba0_dictionary_string`. This data structure implements generic dictionaries which map strings to other objects. In this implementation, the values associated to the keys are integers, which are supposed to be indices in some table of objects of unspecified type. It is implemented using the double hash strategy. The entries of the field `area` contain either -1 or the index, in some unspecified table, of some object. The size of `area` is a power of two given by `log2_size`. This type declaration belongs to `ba0_dictionary_string.h`.

```
struct ba0_dictionary_string
{
// each entry contains either -1 or an index in some table of variables
  struct ba0_tableof_int_p area;
// the base-2 logarithm of the size/alloc field of area if area is nonempty
  ba0_int_p log2_size;
// the number of used entries in area
  ba0_int_p used_entries;
// a function which associates an identifier to an object
```

```
  char *(
      *object_to_ident) (
      void *);
};
```

`void ba0_init_dictionary_string (struct ba0_dictionary_string *D, char *(*object_to_ident) (void *), ba0_int_p log2_size)` [Function]

Initialize D to the empty dictionary. The argument *object_to_ident* points to a function which associates to any element of the dictionary its string identifier. The argument *log2_size* provides the initial size of the field `area` of D. It should be at least equal to 8. This function belongs to `ba0_dictionary_string.c`. It is run by 500 tests, including 6 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_dictionary_string (struct ba0_dictionary_string *D)` [Function]

Reset D to the empty dictionary. This function belongs to `ba0_dictionary_string.c`. It is run by 490 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_dictionary_string (struct ba0_dictionary_string *D, enum ba0_garbage_code code)` [Function]

Return the size needed to perform a copy of D. If *code* is `ba0_embedded` then D is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `ba0_dictionary_string.c`. It is run by 75 tests, including 1 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_dictionary_string (struct ba0_dictionary_string *dst, struct ba0_dictionary_string *src)` [Function]

Assign *src* to *dst*. This function belongs to `ba0_dictionary_string.c`. It is run by 166 tests, including 3 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`ba0_int_p ba0_get_dictionary_string (struct ba0_dictionary_string *D, struct ba0_table *T, char *ident)` [Function]

Return the index in T of the element with identifier *ident*. The element must have been recorded in D. Return `BA0_NOT_AN_INDEX` if not found. This function belongs to `ba0_dictionary_string.c`. It is run by 492 tests, including 5 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_add_dictionary_string (struct ba0_dictionary_string *D, struct ba0_table *T, char *ident, ba0_int_p z)` [Function]

Add to D an entry for the zth element of T which has identifier *ident*. The element may actually not yet be present in T. Exception `BA0_ERRKEY` is raised if the key is already present in D. This function belongs to `ba0_dictionary_string.c`. It is run by 493 tests, including 6 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

4.16.4 Dictionaries with typed string keys

Description of type `ba0_dictionary_typed_string`. Variant of the data structure `ba0_dictionary_string` where keys are couples formed by a string and a type rather than simply a string. This type declaration belongs to `ba0_dictionary_typed_string.h`.

```
struct ba0_dictionary_typed_string
{
// each entry contains either -1 or an index in some table of variables
  struct ba0_tableof_int_p area;
// the base-2 logarithm of the size/alloc field of area if area is nonempty
  ba0_int_p log2_size;
// the number of used entries in area
  ba0_int_p used_entries;
// a function which associates an identifier to an object
  char *(
      *object_to_ident) (
      void *);
// a function which associates a type to an object
    ba0_int_p (
      *object_to_type) (
      void *);
};
```

`void ba0_init_dictionary_typed_string (struct ba0_dictionary_typed_string *D, char *(*object_to_ident) (void *), ba0_int_p (*object_to_type) (void *), ba0_int_p log2_size)` [Function]

Initialize D to the empty dictionary. The argument *object_to_ident* points to a function which associates to any element of the dictionary its string identifier. The argument *log2_size* provides the initial size of the field `area` of D. It should be at least equal to 8. This function belongs to `ba0_dictionary_typed_string.c`. It is run by 490 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_reset_dictionary_typed_string (struct ba0_dictionary_typed_string *D)` [Function]

Reset D to the empty dictionary. This function belongs to `ba0_dictionary_typed_string.c`. It is run by 490 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p ba0_sizeof_dictionary_typed_string (struct ba0_dictionary_typed_string *D, enum ba0_garbage_code code)` [Function]

Return the size needed to perform a copy of D. If *code* is `ba0_embedded` then D is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `ba0_dictionary_typed_string.c`. It is run by 74 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`void ba0_set_dictionary_typed_string (struct ba0_dictionary_typed_string *dst, struct ba0_dictionary_typed_string *src)` [Function]

Assign *src* to *dst*. This function belongs to `ba0_dictionary_typed_string.c`. It is run by 331 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

`ba0_int_p ba0_get_dictionary_typed_string (struct ba0_dictionary_typed_string *D, struct ba0_table *T, char *ident, ba0_int_p type)` [Function]

Return the index in T of the element with identifier *ident* and type *type*. The element must have been recorded in D. Return `BA0_NOT_AN_INDEX` if not found. This function belongs to `ba0_dictionary_typed_string.c`. It is run by 470 tests, including 0 in the `ba0` library and 64 in the `DifferentialAlgebra` package.

`void ba0_add_dictionary_typed_string (struct ba0_dictionary_typed_string *D, struct ba0_table *T, char *ident, ba0_int_p type, ba0_int_p z)` [Function]

Add to D an entry for the zth element of T which has identifier *ident* and type *type*. The element may actually not yet be present in T. Exception `BA0_ERRKEY` is raised if the key is already present in D. This function belongs to `ba0_dictionary_typed_string.c`. It is run by 487 tests, including 0 in the `ba0` library and 65 in the `DifferentialAlgebra` package.

5 The bav library

5.1 Introduction

The bav library implements functionalities for manipulating the variables which are used at higher level to build polynomials. It is specialized for differential algebra. The key feature is an implementation of *orderings* on variables. Orderings contain the so called *rankings* but not only. It also implement *terms* i.e. products of powers of variables.

The bav library relies on the ba0 library.

5.1.1 Settings

Read the documentation of the ba0 library about settings.

`void bav_reset_all_settings (void)` [Function]
Reset all settings variables to their default values. This function must be called outside sequences of calls to the library. This function belongs to `bav_common.c`. It is run by 65 tests, including 0 in the bav library and 65 in the `DifferentialAlgebra` package.

5.1.2 Basic types, constants and naming conventions

The name of every exported object defined in the library starts by `bav_`. Functions starting by `bav_init` initialize raw data structures (constructors). Functions starting by `bav_new` dynamically allocate data structures and apply the corresponding `bav_init` functions on them. Functions starting by `bav_reset` reset data structures which are assumed to be already initialized. Functions starting by `bav_set` perform assignments and systematically copy the sources to the destinations.

The four integer types `bav_Idegree`, `bav_Iordering`, `bav_Inumber` and `bav_Iorder` are redefinitions of `ba0_int_p`. They were introduced for legibility.

Every sequence of calls to the bav library should start with a call to `bav_restart` and finish with a call to `bav_terminate`.

`void bav_restart (ba0_int_p` *`time_limit`*`, ba0_int_p` *`memory_limit`*`)` [Function]
Call `ba0_restart` with the same parameters. Define a few more formats. Push the address of `bav_global.R.ord_stack.size` which defines the so called *current ordering* in the exception extra stack. This function belongs to `bav_common.c`. It is run by 494 tests, including 102 in the bav library and 65 in the `DifferentialAlgebra` package.

`void bav_terminate (enum ba0_restart_level` *`level`*`)` [Function]
Call `ba0_terminate` with the same parameter. This function belongs to `bav_common.c`. It is run by 494 tests, including 102 in the bav library and 65 in the `DifferentialAlgebra` package.

5.2 Global variables and settings

Setting variables are global configuration variables which permit to tune the behaviour of the library functions. This tuning is supposed to be performed before `bav_restart` is called (more generally, outside any sequence of calls to the library). They are bundled in the `bav_initialized_global` global variable which has the following type.

Description of type `bav_initialized_global`. This type declaration belongs to `bav_global.h`.

```
struct bav_initialized_global
{
  struct
  {
/*
 * Function called when an unknown symbol/variable/parameter is parsed.
 * Default bav_unknown_default
 */
    ba0_indexed_string_function *unknown;
  } common;
  struct
  {
/*
 * Functions pointers for customizing symbol parsing and printing
 */
    ba0_scanf_function *scanf;
    ba0_printf_function *printf;
/*
 * If IndexedBase_PFE is not (char *)0, the radical z of every symbol
 * z[something] which fits some range indexed group is printed
 * IndexedBase_PFE('z') leading to a symbol IndexedBase_PFE('z')[something].
 * This mechanism prevents some evaluation failures in Python. Default 0
 */
    char *IndexedBase_PFE;
  } symbol;
  struct
  {
/*
 * If Function_PFE is not (char *)0, every parameter p with nonempty
 * dependencies is printed Function_PFE('p')(dependencies).
 * This mechanism prevents some evaluation failures in Python. Default 0.
 */
    char *Function_PFE;
  } parameter;
  struct
  {
/*
 * Functions pointers for customizing variable parsing and printing
 */
    ba0_scanf_function *scanf;
    ba0_printf_function *printf;
/*
 * The strings which stand for no derivation in the jet0 notation
 */
    char *jet0_input_string;
    char *jet0_output_string;
/*
 * The prefix of temporary variables
 */
```

```
    char *temp_string;
  } variable;
  struct
  {
/*
 * Function pointer for customizing the way ranks are printed.
 */
    ba0_printf_function *printf;
  } rank;
  struct
  {
/*
 * The string for displaying orderings
 */
    char *string;
  } ordering;
};
```

Some global variables are read and modified during sequences of calls to the library. They are bundled in the `bav_global` variable which has the following type.

Description of type `bav_global`. This type declaration belongs to `bav_global.h`.

```
struct bav_global
{
  struct
  {
/*
 * Receives the faulty string when an unknown variable/symbol is parsed
 */
    char unknown[BA0_BUFSIZE];
  } common;
  struct bav_differential_ring R;
  struct
  {
/*
 * Flags indicating which input notation was used
 */
    ba0_int_p notations;
/*
 * "diff" or "Diff" to display derivatives
 */
    char *diff_string;
  } variable;
  struct
  {
/*
 * Comparison functions w.r.t. term orderings
 */
    enum ba0_compare_code (
        *compare) (
        struct bav_term *,
        struct bav_term *);
    enum ba0_compare_code (
        *compare_stripped) (
        struct bav_term *,
        struct bav_term *,
        bav_Inumber);
  } term_ordering;
};
```

`void bav_set_settings_common (ba0_indexed_string_function *u)` [Function]

Set `bav_custom_unknown` to u if it is nonzero else set it to `bav_unknown_default`. This function belongs to `bav_common.c`. It is run by 65 tests, including 0 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_get_settings_common (ba0_indexed_string_function **u)` [Function]

Set u to `bav_custom_unknown`. The parameter u may be zero. This function belongs to `bav_common.c`. It is not run by any test.

Description of type `bav_PFE_settings`. This data structure is used to store copies of the settings variables contained in the fields of `bav_initialized_global` which contain `PFE` variables. This type declaration belongs to `bav_global.h`.

```
struct bav_PFE_settings
{
  struct ba0_PFE_settings ba0;
  ba0_scanf_function *scanf;
  ba0_printf_function *printf;
  char *IndexedBase_PFE;
  char *Function_PFE;
};
```

`void bav_cancel_PFE_settings (struct bav_PFE_settings *P)` [Function]

Fill the fields of P with the corresponding settings variables. Then cancel the `PFE` settings. This function belongs to `bav_common.c`. It is run by 2 tests, including 0 in the `bav` library and 2 in the `DifferentialAlgebra` package.

`void bav_restore_PFE_settings (struct bav_PFE_settings *P)` [Function]

Restore the `PFE` settings with the content of P. This function belongs to `bav_common.c`. It is run by 2 tests, including 0 in the `bav` library and 2 in the `DifferentialAlgebra` package.

5.3 Differential rings

Description of type `bav_differential_ring`. This data type implements one mathematical differential ring endowed with many different orderings. It is completely stored in `ba0_global.stack.quiet` (the quiet stack).

The field `strs` contains all the strings used as identifiers of symbols or as radicals of range indexed strings. Note 1: identifiers of symbols involve identifiers of variables which fit range indexed strings. Note 2: the same string may be both the radical of a range indexed string and the identifier of a differential indeterminate.

The `syms` table contains all the symbols. The `vars` table contains all the variables.

The dictionary `dict_str_to_var` maps strings to variables of the `vars` table. Only the radical variables are stored in this dictionary.

The dictionary `dict_str_to_rig` maps strings to range indexed strings of the `rigs` table. The keys are the radicals of the range indexed strings.

The table `ders` contains the indices in `vars` of the derivations / independent variables. The index in `ders` of a derivation is called its *derivation index*. This table thus maps derivation indices to the corresponding independent variables.

The field `ords` contains all the defined orderings. They all refer to the same mathematical differential ring hence must order the same set of symbols. It actually associates an index to each ordering and permits to interpret the table `number` of the variables.

The field `ord_stack` implements a stack of indices in `ords`. Its top element is the *current ordering*.

The field `pars` contains the description of the parameters of the differential ring i.e. dependent symbols some derivatives of which simplifying to zero.

Many functions of the `bav` library hold identifiers starting with `bav_R`. These are low level functions, performing critical operations of the `bav_global.R` structure. This type declaration belongs to `bav_differential_ring.h`.

```
struct bav_differential_ring
{
// true if the differential ring is empty
  bool empty;
// the table of all ident of symbols, radicals of range indexed strings
  struct ba0_tableof_string strs;
// the table of the range indexed strings
  struct ba0_tableof_range_indexed_group rigs;
// the table of symbols
  struct bav_tableof_symbol syms;
// the table of variables
  struct bav_tableof_variable vars;
// maps strings to strings in strs
  struct ba0_dictionary_string dict_str_to_str;
// maps strings to variables in vars
  struct ba0_dictionary_string dict_str_to_var;
// maps strings (radicals) to range indexed strings in rigs
  struct ba0_dictionary_string dict_str_to_rig;
// the indices in vars of the derivations
  struct ba0_tableof_int_p ders;
// the indices in vars of the temporary variables
  struct ba0_tableof_int_p tmps;
// tmps_in_use[i] is nonzero iff tmps[i] is in use
  struct ba0_tableof_int_p tmps_in_use;
//  -1 or the index in vars of the symbol used for differential operators
  ba0_int_p opra;
// all the defined orderings
  struct bav_tableof_ordering ords;
// a stack of indices in ords
  struct ba0_tableof_int_p ord_stack;
// the parameters
  struct bav_parameters pars;
};
```

`void bav_init_differential_ring (struct bav_differential_ring *R)` [Function]

Initialize R to the empty differential ring. This function belongs to `bav_differential_ring.c`. It is run by 494 tests, including 102 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_differential_ring * bav_new_differential_ring (void)` [Function]

Allocate a new differential ring and initialize it. This function belongs to `bav_differential_ring.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`bool bav_is_empty_differential_ring (struct bav_differential_ring *R)` [Function]

Return `true` if R is empty. This function belongs to `bav_differential_ring.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_differential_ring (struct bav_differential_ring *R, enum ba0_garbage_code code)` [Function]

Return the overall size of R: the one which is needed by `bav_R_set_differential_ring` to perform a copy of R. If *code* is `ba0_embedded` then R is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `bav_differential_ring.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_differential_ring (struct bav_differential_ring *R, struct bav_differential_ring *S)` [Function]

Copy S into R. The strings, symbols and variables, which are not duplicated in S, do not get duplicated in R either. The needed size for the result is the one returned by `bav_sizeof_differential_ring`. This function is called from the `bmi` library. This function belongs to `bav_differential_ring.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_create_differential_ring (struct ba0_tableof_string *D, struct bav_tableof_block *B, struct bav_block *O)` [Function]

Create in the quiet stack the mathematical differential polynomial ring over which all orderings will be defined using D (the derivation list), B (the block list) and O (the operator). This function is called when parsing the first ordering in a sequence of calls to the library but does not define the ordering: this task is left to a following call to `bav_R_new_ordering`. All symbol identifiers occurring in D, B and O lead to new symbols and variables. Exception `BA0_ERRALG` is raised if `bav_global.R` is not empty. Exceptions `BAV_ERRBOR` or `BAV_ERRDSY` are raised in case of a bad ordering. This function belongs to `bav_differential_ring.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`bool bav_R_ambiguous_symbols (void)` [Function]

Return `true` if one of the symbols present in `bav_global.R.syms` is an indexed string terminated by an indexed string indices made of a sequence of independent symbols. This function belongs to `bav_differential_ring.c`. It is run by 67 tests, including 2 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_R_new_symbol_and_variable (char *ident, enum bav_typeof_symbol type, ba0_int_p index_in_rigs, struct ba0_tableof_int_p *subscripts)` [Function]

This low level function creates a new symbol and the corresponding new variable using *ident*, *type*, *index_in_rigs* and *subscripts*. If *index_in_rigs* is not `BA0_NOT_AN_INDEX` then the trailing indexed string indices of *ident* forms a sequence of integers, which is stored in numeric form in *subscripts*. Otherwise, *subscripts* may be zero. The function is called from the parsers and by `bav_R_new_temporary_variable`. This function belongs to `bav_differential_ring.c`. It is run by 136 tests, including 30 in the `bav` library and 13 in the `DifferentialAlgebra` package.

5.4 Orderings

Description of type `bav_ordering`. This data type is a subtype of `bav_differential_ring` used to describe orderings. In differential algebra, derivatives of differential indeterminates are ordered with respect to a *ranking*. In the software, the notion of ranking is generalized to the one of *ordering*, to take into account independent variables, temporary variables or derivatives temporarily considered as maximal or minimal variables. The simplest method to define an ordering in the `bav` library consists in using the parser. Let us thus have a look to a few different definitions of orderings, accepted by the parser.

```
1. ordering (derivations = [t], blocks = [[u,v], grlexB[w]])
2. ordering (derivations = [], blocks = [w,v,u])
3. ordering (derivations = [x,y], blocks = [degrevlexA[u]],
operator = [D], varmax = [u[x,x]])
4. ordering (derivations = [x,y], blocks = [degrevlexB[u],v],
varmax = [u[x],u[x,x,x]], varmin = [u[x,x]])
```

An ordering is first defined by a list of blocks $[b_1, \ldots, b_n]$. Leftmost blocks are considered as greater than rightmost ones i.e. every derivative of any differential indeterminate present in b_i is greater than any derivative of any differential indeterminate present in b_{i+k} (with $k > 0$). In Example 1, we have $v_t > w_{t,t,t}$. In Example 2, we have $w > v > u$.

Within a fixed block, derivatives are ordered with respect to a *subranking* (see the description of the data type `bav_subranking`). The default subranking is `grlexA`. Other subrankings such as `grlexB`, `degrevlexA` or `degrevlexB` appear in the above examples. Subrankings are only meaningful in the partial case. In the ordinary case everything simplifies as follows. Consider a block $[u_1, \ldots, u_n]$ of differential indeterminates. Derivatives of the u_i are first compared with respect to their *order*. In the case of two derivatives with the same order, leftmost differential indeterminates are considered as greater than rightmost ones.

Independent variables / derivations are considered as lower than derivatives. In the partial case, leftmost derivations are considered as greater than rightmost ones, in the list of derivations. In Examples 3 and 4, we have $x > y$.

The orderings defined so far mostly are rankings (slightly generalized to cover the case of independent variables). They are suitable for differential elimination algorithms. However, it is also sometimes useful to use orderings which are not rankings by specifying variables considered as greater than or lower than any other variable (such orderings should only be used when no differentiation process is involved). This is achieved by means of the `varmax`

and the `varmin` lists. In Example 3, $u_{x,x}$ is greater than any other variable. In Example 4, u_x and $u_{x,x,x}$ are greater than any other variable. Moreover $u_x > u_{x,x,x}$ and $u_{x,x}$ is lower than any other variable.

In principle, it is also possible to define a *differential operator*. Derivatives of the differential operator are ordered as variables, using a block description. Any derivative of the differential operator is greater than any derivative of any differential indeterminate. However, derivatives listed in the `varmax` list are considered as greater than derivatives of the differential operator.

It is moreover possible to define temporary variables. These variables are considered as lower than any other variable. Among themselves, temporary variables are ordered according to some session dependent unspecified ordering. These temporary variables are handled in the `bav_differential_ring` data structure.

The fields `typed_ident` and `dict` permit to associate a triple (i, j, k) of indices to a typed identifier *ident*. They are used for computing variable numbers. The first index i is the block number of *ident* in the sense that *ident* belongs to the ith entry of the `blocks` table. The second index j is the index of *ident* within the ith block. The third index k is is meaningful in the case of range indexed groups which are not plain strings. It gives the index of *ident* in the group. Consider the following example, featuring a table of two blocks.

```
blocks = [[z], [(y,z)[inf:-1], t, (u)[3:5]]]
```

Then the `typed_ident` field contains the three following entries:

```
typed_ident = [
['z', bav_plain_ident, 0, 0, -1],
['y', bav_range_indexed_string_radical_ident, 1, 0, 0],
['z', bav_range_indexed_string_radical_ident, 1, 0, 1],
['t', bav_plain_ident, 1, 1, -1],
['u', bav_range_indexed_string_radical_ident, 1, 2, 0]]
```

This type declaration belongs to `bav_ordering.h`.

```
struct bav_ordering
{
// the table of derivations - the order matters for some subrankings
  struct bav_tableof_symbol ders;
// the table of blocks
  struct bav_tableof_block blocks;
// the array of typed ident and its associated dictionary.
  struct bav_tableof_typed_ident typed_idents;
  struct ba0_dictionary_typed_string dict;
// the block for the differential operator - if any
  struct bav_block operator_block;
// the variables set as maximal variables
  struct bav_tableof_variable varmax;
// the variables set as minimal variables
  struct bav_tableof_variable varmin;
};
```

Description of type `bav_tableof_ordering`. This type declaration belongs to `bav_ordering.h`.

```
struct bav_tableof_ordering
{
  ba0_int_p alloc;
  ba0_int_p size;
```

```
  struct bav_ordering **tab;
};
```

`void bav_set_settings_ordering (char *ordstring)` [Function]
Set to *ordstring* the leading string used for parsing and printing orderings. If zero, this leading string is reset to its default value: `ordering`. This function belongs to `bav_ordering.c`. It is run by 66 tests, including 1 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_get_settings_ordering (char **ordstring)` [Function]
Assign to *ordstring* the leading string used for printing orderings. This function belongs to `bav_ordering.c`. It is not run by any test.

`unsigned ba0_int_p bav_sizeof_ordering (struct bav_ordering *ord, enum ba0_garbage_code code, bool strings_syms_and_vars_not_copied)` [Function]
Return the size needed to perform a copy of *ord*. If *code* is `ba0_embedded` then *ord* is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_syms_and_vars_not_copied* is `true` then the strings, symbols and variables occurring in *ord* are supposed not to be copied. This function belongs to `bav_ordering.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_tableof_ordering (struct bav_tableof_ordering *T, enum ba0_garbage_code code, bool strings_syms_and_vars_not_copied)` [Function]
Return the size needed to perform a copy of T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_syms_and_vars_not_copied* is `true` then the strings, symbols and variables occurring in the elements of T are supposed not to be copied. This function belongs to `bav_ordering.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_init_ordering (struct bav_ordering *o)` [Function]
Initialize *o* to the empty ordering. This function belongs to `bav_ordering.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_reset_ordering (struct bav_ordering *o)` [Function]
Empty the ordering *o*. This function belongs to `bav_ordering.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_ordering * bav_new_ordering (void)` [Function]
Allocate a new ordering in the current stack, initialize it and return it. This function belongs to `bav_ordering.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`bool bav_has_varmax_current_ordering (void)` [Function]
Return `true` if the current ordering involves a `varmax` definition. This function belongs to `bav_ordering.c`. It is run by 191 tests, including 0 in the `bav` library and 32 in the `DifferentialAlgebra` package.

`void bav_set_ordering (struct bav_ordering *R, struct bav_ordering *S)` [Function]

Assign S to R. The two orderings share their `blocks` field. This function belongs to `bav_ordering.c`. It is run by 298 tests, including 1 in the `bav` library and 41 in the `DifferentialAlgebra` package.

`void * bav_scanf_ordering (void *z)` [Function]

The general parsing function for orderings. It can be called by `ba0_scanf/%ordering`. Parsing the first ordering creates the mathematical differential ring. The parsing grammar is the following.

```
ORDERING ::= START-KEYWORD "(" [FIRST-GROUP] ["," SECOND-GROUP] ")"
START-KEYWORD ::= ordering | DifferentialRing
FIRST-GROUP   ::= derivations = %t[%six] ["," FIRST_GROUP]
::= blocks = %t[%b] ["," FIRST_GROUP]
::= operator = %b ["," FIRST_GROUP]
SECOND-GROUP  ::= varmax = %t[%v] ["," SECOND-GROUP]
::= varmin = %t[%v] ["," SECOND-GROUP]
::= parameters = %t[%param] ["," SECOND-GROUP]
```

Each keyword of each group must occur at most once.

The `parameters = [...]` argument permits to define the parameters of the mathematical differential ring. Though it is slightly inconsistent to define parameters with orderings, their definition has been inserted here for a tighter relationship with the user interfaces of the `DifferentialAlgebra` packages. This argument is used only when creating the first ordering (hence the mathematical differential ring). For further orderings, it is ignored and needs not be provided.

Exception `BAV_ERRBOR` is raised in case of an error. This function belongs to `bav_ordering.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_printf_ordering (void *z)` [Function]

The printing function for orderings. It is called by `ba0_printf/%ordering`. This function belongs to `bav_ordering.c`. It is run by 3 tests, including 0 in the `bav` library and 3 in the `DifferentialAlgebra` package.

`bav_Iordering bav_R_new_ranking (struct ba0_tableof_string *D, struct bav_tableof_block *B, struct bav_block *O)` [Function]

This function belongs to `bav_differential_ring.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_add_block_to_all_orderings (struct bav_block *B, ba0_int_p block_number)` [Function]

Add the block B to all the orderings of `bav_global.R` with block number equal to *block_number*. The block number is restricted to the values 0 (the new block is higher than any other block) or -1 (the new block is lower than any other block). This function belongs to `bav_differential_ring.c`. It is run by 6 tests, including 2 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`bav_Iordering bav_R_copy_ordering (bav_Iordering r)` [Function]

Duplicate an already existing ordering and return its identifier. The two orderings share their `blocks` field. This function belongs to `bav_differential_ring.c`. It is run by 298 tests, including 1 in the `bav` library and 41 in the `DifferentialAlgebra` package.

`void bav_R_swap_ordering (bav_Iordering r, bav_Iordering rbar)` [Function]

Swap the orderings *r* and *rbar*. This function may be useful to free an ordering which is not the last created one (see `bav_R_free_ordering`). This function belongs to `bav_differential_ring.c`. It is not run by any test.

`void bav_R_free_ordering (bav_Iordering r)` [Function]

Free the ordering *r*. This function requires that *r* is the index of the last created ordering. This function belongs to `bav_differential_ring.c`. It is run by 296 tests, including 1 in the `bav` library and 43 in the `DifferentialAlgebra` package.

`void bav_R_set_ordering (struct bav_ordering *dst, struct bav_ordering *src, struct bav_differential_ring *R)` [Function]

Copy *src* into *dst* without performing any string allocation. The strings present in *src* are supposed to have copies in *R*. Instead of duplicating them, the copies are used. The needed size for the result is the one returned by `bav_sizeof_ordering`. This function belongs to `bav_ordering.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_tableof_ordering (struct bav_tableof_ordering *dst, struct bav_tableof_ordering *src, struct bav_differential_ring *R)` [Function]

Copy *src* into *dst* by calling `bav_R_set_ordering`. See this function. This function belongs to `bav_ordering.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`int bav_R_compare_variable (struct bav_variable *v, struct bav_variable *w)` [Function]

Return -1, 0 or 1 if $v < w$, $v = w$ or $v > w$ with respect to the current ordering. This low level function should only be used when the numbers of the variables with respect to the current ordering are not yet computed. This function belongs to `bav_ordering.c`. It is run by 475 tests, including 97 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_sort_tableof_variable (struct bav_tableof_variable *T)` [Function]

Sort T by increasing order with respect to the current ordering. This low level function should only be used when the numbers of the variables with respect to the current ordering are not yet computed. This function belongs to `bav_ordering.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_lower_block_range_indexed_strings (struct ba0_tableof_range_indexed_group *G, ba0_int_p first_index)` [Function]

This function belongs to `bav_ordering.c`. It is run by 66 tests, including 0 in the `bav` library and 5 in the `DifferentialAlgebra` package.

5.5 Current ordering

`void bav_push_ordering (bav_Iordering r)` [Function]

Push r on the top of the ordering stack `bav_global.R.ord_stack`. The ordering r becomes the current ordering. This function belongs to `bav_differential_ring.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_pull_ordering (void)` [Function]

Undo the last call to `bav_push_ordering`. This function belongs to `bav_differential_ring.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`bav_Inumber bav_current_ordering (void)` [Function]

Return the current ordering. This function belongs to `bav_differential_ring.c`. It is run by 488 tests, including 99 in the `bav` library and 65 in the `DifferentialAlgebra` package.

5.6 Typed identifiers

Description of type `bav_typeof_ident`. This data type is a subtype of `bav_typed_ident`. This type declaration belongs to `bav_typed_ident.h`.

```
enum bav_typeof_ident
{
// The identifier is a plain string
  bav_plain_ident,
// The identifier is the radical of a range indexed string
  bav_range_indexed_string_radical_ident
};
```

Description of type `bav_typed_ident`. This data structure permits to associate a table of indices to typed identifiers. As a substructure of `bav_ordering`, it associates to any typed identifier three indices: the index of the block which contains the identifier; the index of the identifier within the block; the index of the identifier in the group (this last index is only meaningful if the type is `bav_range_indexed_string_radical_ident`). This type declaration belongs to `bav_typed_ident.h`.

```
struct bav_typed_ident
{
// the string - possibly (char *)0
  char *ident;
// its type
  enum bav_typeof_ident type;
// the indices associated to the typed ident
  struct ba0_tableof_int_p indices;
};
```

`void bav_init_typed_ident (struct bav_typed_ident *tid)` [Function]
Initialize *tid* to the empty typed ident. This function belongs to `bav_typed_ident.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_typed_ident * bav_new_typed_ident (void)` [Function]
Allocate a new typed ident, initialize it and return it. This function belongs to `bav_typed_ident.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`ba0_int_p bav_get_typed_ident_from_symbol (struct ba0_dictionary_typed_string *D, struct bav_tableof_typed_ident *T, struct bav_symbol *y)` [Function]
Return the index in T of the typed ident associated to y. The typed ident must have been recorded in D. Return `BA0_NOT_AN_INDEX` if not found. This function belongs to `bav_typed_ident.c`. It is run by 470 tests, including 97 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_typed_ident (struct bav_typed_ident *tid, enum ba0_garbage_code code, bool strings_not_copied)` [Function]
Return the size needed to perform a copy of *tid*. If *code* is `ba0_embedded` then *tid* is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* is `true` then the `ident` field of *tid* is supposed not to be copied. This function belongs to `bav_typed_ident.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_tableof_typed_ident (struct bav_tableof_typed_ident *T, enum ba0_garbage_code code, bool strings_not_copied)` [Function]
Return the size needed to perform a copy of T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* is `true` then the strings occurring in T are supposed not to be copied. This function belongs to `bav_typed_ident.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_typed_ident (struct bav_typed_ident *dst, struct bav_typed_ident *src, struct bav_differential_ring *R)` [Function]
Copy *src* into *dst* without performing any string allocation. The strings present in *src* are supposed to have copies in R. Instead of duplicating them, the copies are used. The needed size for the result is the one returned by `bav_sizeof_typed_ident` with *strings_not_copied* set to `true`. This function belongs to `bav_typed_ident.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_tableof_typed_ident (struct bav_tableof_typed_ident *dst, struct bav_tableof_typed_ident *src, struct bav_differential_ring *R)` [Function]

Copy *src* into *dst* by calling `bav_R_set_typed_ident`. See this function. This function belongs to `bav_typed_ident.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_append_tableof_typed_ident_block (struct bav_tableof_typed_ident *T, ba0_int_p first_index, struct bav_block *B, struct bav_differential_ring *R)` [Function]

Append to T the entries corresponding to the block B with first index field equal to *first_index*. The strings present in B are supposed to have copies in R. Instead of duplicating them, the copies are used. This function belongs to `bav_typed_ident.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_tableof_typed_ident_tableof_block (struct bav_tableof_typed_ident *T, struct bav_tableof_block *B, struct bav_differential_ring *R)` [Function]

Fill the table T with all the blocks occurring in B. The strings present in B are supposed to have copies in R. Instead of duplicating them, the copies are used. This function belongs to `bav_typed_ident.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

5.7 Blocks

Description of type `bav_block`. A *block* is a list of range indexed groups defining differential indeterminates (possibly also a single differential operator) ordered with respect to some common subranking. See the description of the data types `bav_subranking` and `bav_ordering`. This type declaration belongs to `bav_block.h`.

```
struct bav_block
{
// the subranking which applies to the block
  struct bav_subranking *subr;
// the table of the range indexed groups of the block
  struct ba0_tableof_range_indexed_group rigs;
};
```

`unsigned ba0_int_p bav_sizeof_block (struct bav_block *b, enum ba0_garbage_code code, bool strings_not_copied)` [Function]

Return the size needed to copy b. If *code* is `ba0_embedded` then b is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* then the strings occurring in b are supposed not to be copied. This function belongs to `bav_block.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_tableof_block (struct bav_tableof_block *T, enum ba0_garbage_code code, bool strings_not_copied)` [Function]

Return the size needed to copy T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* then the strings occurring in the blocks of T are supposed not to be copied. This function belongs to `bav_block.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_init_block (struct bav_block *b)` [Function]

Initialize b to the empty block. This function belongs to `bav_block.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_block * bav_new_block (void)` [Function]

Allocate a new block in the current stack, initialize it and return it. This function belongs to `bav_block.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_reset_block (struct bav_block *b)` [Function]

Empty the block b. This function belongs to `bav_block.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`bool bav_is_empty_block (struct bav_block *b)` [Function]

Return `true` if b is empty. This function belongs to `bav_block.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void * bav_scanf_block (void *z)` [Function]

The parsing function for blocks. It is called by `ba0_scanf/%b`. Exception `BAV_ERRBLO` can be raised. This function belongs to `bav_block.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_printf_block (void *z)` [Function]

The printing function for blocks. It is called by `ba0_printf/%b`. This function belongs to `bav_block.c`. It is run by 3 tests, including 0 in the `bav` library and 3 in the `DifferentialAlgebra` package.

`void bav_R_set_block (struct bav_block *dst, struct bav_block *src, struct bav_differential_ring *R)` [Function]

Copy *src* into *dst* without performing any string allocation. The strings occurring in *src* are supposed to be present in R. Each time a string has to be assigned, its copy in R is used. The size needed for the copy is the one returned by `bav_sizeof_block`. This function belongs to `bav_block.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_tableof_block (struct bav_tableof_block *dst, struct bav_tableof_block *src, struct bav_differential_ring *R)` [Function]

Copy *src* into *dst* without performing any string allocation by using `bav_R_set_block`. See this function. This function belongs to `bav_block.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

5.8 Subrankings

Description of type `bav_subranking`. This data type implements subrankings. A *subranking* permits to compare derivatives of differential indeterminates. Subrankings form a subtype of the `bav_block` data type.

There are five predefined subrankings: `lex`, `grlexA`, `grlexB`, `degrevlexA` and `degrevlexB`. The difference between them is actually only meaningful in the case of partial derivatives. Consider two different derivatives $u = \theta\, y$ and $v = \phi\, z$ to be compared. View the *derivative operators* θ, ϕ as power products of derivations.

Among themselves, differential indeterminates are ordered by their order of appearance in the block. Among themselves, derivations are ordered by their order of appearance in the derivation list: this permits to compare derivative operators. For details, see the `bav_ordering` data structure.

In the `lex` ordering, derivative operators are first compared with respect to the lexicographic order; in case of equality, differential indeterminates are compared.

In all other subrankings, derivative operators are first compared with respect to their total order. In the sequel, it is assumed that θ and ϕ have the same total order.

In the `grlexA` and `degrevlexA` subrankings, the second comparison criterion is given by the differential indeterminates i.e. $u > v$ if $y > z$. In the case of equality, the derivative operators are compared either with respect to the lexicographic ordering (`grlexA`) or with respect to the reverse lexicographic ordering (`degrevlexA`).

In the `grlexB` and `degrevlexB` subrankings, the second comparison criterion is given by the derivative operators, which are compared either with respect to the lexicographic ordering (`grlexB`) or with respect to the reverse lexicographic ordering (`degrevlexB`). In the case of equality, the differential indeterminates are compared.

To each subranking, a comparison function `inf` is associated. This function is called when a new ordering or a new variable are created. It is involved in the comparison process of two variables, in order to determine their numbers, with respect to some ordering *ord*.

This function takes as input two different variables v and w belonging to some same block of *ord*, their typed ident *tid_v* and *tid_w*, picked from the `typed_idents` field of *ord* and the address of the `ders` field of *ord*. It returns `true` if $v < w$ with respect to the subranking (hence the ordering), `false` otherwise. This type declaration belongs to `bav_subranking.h`.

```
struct bav_subranking
{
// The identifier of the subranking (grlexA, degrevlexB, ...)
// This identifier is a readonly global string.
  char *ident;
  bool (
      *inf) (
      struct bav_variable * v,
      struct bav_variable * w,
      struct bav_typed_ident * tid_v,
      struct bav_typed_ident * tid_w,
      struct bav_tableof_symbol * ders);
};
```

`bool bav_is_subranking (char *ident, struct bav_subranking **subranking)` [Function]

Return `true` if *ident* is the identifier of a subranking. If so, *subranking* is assigned the corresponding value, read from a static readonly array. This function belongs to `bav_subranking.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

5.9 Symbols

Description of type `bav_symbol`. A *symbol* mostly is an identifier endowed with a type. Symbols are used to define variables. In the case of an *independent symbol* or a *temporary symbol*, the symbol is associated to a unique corresponding variable. In the case of a *dependent symbol* or an *operator symbol*, the symbol is shared by many different variables, corresponding to its derivatives.

Symbols are not duplicated. They are allocated in `ba0_global.stack.quiet` and stored in the table `bav_global.R.syms`.

A symbol may be a *plain symbol* or a symbol which fits a range indexed string. The latter case can only occur if the symbol is dependent.

If the symbol is a plain symbol then its field `index_in_rigs` contains `BA0_NOT_AN_INDEX` and the field `subscripts` is empty. If it fits a range indexed string then its field `index_in_rigs` contains the index in `bav_global.R.rigs` of the radical of its identifier and the field `subscripts` contains its subscripts.

A symbol may be a parameter or not. Only dependent symbols may be parameters. If the symbol is a parameter then its field `index_in_pars` contains the index in `bav_global.pars.pars` of its parameter description. If the symbol is not a parameter, its field `index_in_pars` contains `BA0_NOT_AN_INDEX`. This type declaration belongs to `bav_symbol.h`.

```
struct bav_symbol
{
// the identifier of the symbol
  char *ident;
// its type
  enum bav_typeof_symbol type;
// the index of the symbol in bav_global.R.syms
  ba0_int_p index_in_syms;
// if ident fits a range indexed string, an index in bav_global.R.rigs
  ba0_int_p index_in_rigs;
// if ident fits a range indexed string, the table of its subscripts
  struct ba0_tableof_int_p subscripts;
// if the symbol is independent, its index in bav_global.R.ders
  ba0_int_p derivation_index;
// if the symbol is a parameter, its index in bav_global.R.pars.pars
  ba0_int_p index_in_pars;
};
```

Description of type `bav_typeof_symbol`. Independent symbols are used to build derivations. Dependent symbols are used to build differential indeterminates. Operator symbols are used to build differential operators. Temporary symbols are used to build temporary variables. This type declaration belongs to `bav_symbol.h`.

```
enum bav_typeof_symbol
```

```
{
  bav_independent_symbol,
  bav_dependent_symbol,
  bav_operator_symbol,
  bav_temporary_symbol
};
```

`void bav_init_symbol (struct bav_symbol *y)` [Function]
Initialize *s* to the empty symbol. This function belongs to `bav_symbol.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_symbol * bav_new_symbol (void)` [Function]
Allocate a new symbol, initialize it and return it. This function belongs to `bav_symbol.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_symbol * bav_not_a_symbol (void)` [Function]
Return the constant `BAV_NOT_A_SYMBOL`. This function belongs to `bav_symbol.c`. It is not run by any test.

`unsigned ba0_int_p bav_sizeof_symbol (struct bav_symbol *y, enum ba0_garbage_code code, bool strings_not_copied)` [Function]
Return the size needed to copy *y*. If *code* is `ba0_embedded` then *y* is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* then the strings occurring in *y* are supposed not to be copied. This function belongs to `bav_symbol.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_set_settings_symbol (ba0_scanf_function *scanf_symbol, ba0_printf_function *printf_symbol, char *IndexedBase_PFE)` [Function]
Set to *scanf_symbol* the function which prints symbols. Set to *printf_symbol* the function which prints symbols. Set to *IndexedBase_PFE* the string used to protect the radicals of the symbols which fit range indexed groups when they get printed. If these parameters are zero, the settings variables are reset to their default values. This function belongs to `bav_symbol.c`. It is run by 69 tests, including 0 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_get_settings_symbol (ba0_scanf_function **scanf_symbol, ba0_printf_function **printf_symbol, char **IndexedBase_PFE)` [Function]
Assign to **scanf_symbol* the function which prints symbols. Assign to **printf_symbol* the function which prints symbols. Assign to **IndexedBase_PFE* the string used to protect the radicals of the symbols which fit range indexed groups when they get printed. Each argument may be zero. This function belongs to `bav_symbol.c`. It is run by 69 tests, including 0 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`bool bav_is_a_derivation (char *string)` [Function]
Return `true` if *string* is equal to the `ident` field of an existing derivation. This function belongs to `bav_symbol.c`. It is run by 385 tests, including 86 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`bool bav_is_subscripted_symbol (struct bav_symbol *y)` [Function]
Return `true` if y fits a range indexed group (hence has subscripts) else `false`. This function belongs to `bav_symbol.c`. It is run by 99 tests, including 0 in the `bav` library and 10 in the `DifferentialAlgebra` package.

`ba0_int_p bav_subscript_of_symbol (struct bav_symbol *y)` [Function]
Return the subscript of y, assuming that y fits a range indexed string and has a single subscript. Exception `BA0_ERRALG` is raised if y does not fit any range indexed string. Exception `BA0_ERRNYP` is raised if y has more than one subscript. This function belongs to `bav_symbol.c`. It is run by 99 tests, including 0 in the `bav` library and 10 in the `DifferentialAlgebra` package.

`void bav_set_tableof_tableof_symbol (struct bav_tableof_tableof_symbol *dst, struct bav_tableof_tableof_symbol *src)` [Function]
Assign *src* to *dst*. This function belongs to `bav_symbol.c`. It is not run by any test.

`int bav_compare_symbol (const void *y0, const void *z0)` [Function]
Comparison function for `qsort`. When used by `qsort`, the resulting table is sorted by decreasing order with respect to the current ordering (symbols are identified with their associated variables). This function belongs to `bav_symbol.c`. It is run by 62 tests, including 0 in the `bav` library and 4 in the `DifferentialAlgebra` package.

`struct bav_symbol * bav_switch_ring_symbol (struct bav_symbol *y, struct bav_differential_ring *R)` [Function]
Return the symbol of R which has the same index in `R->syms` as y. This low level function should be used in conjunction with `bav_set_differential_ring`: if R is a ring obtained by application of `bav_set_differential_ring` to the ring y refers to, then this function returns the element of R which corresponds to y. This function belongs to `bav_symbol.c`. It is run by 11 tests, including 0 in the `bav` library and 10 in the `DifferentialAlgebra` package.

`void * bav_scanf_symbol (void *z)` [Function]
The general parsing function for symbols. It is called by `ba0_scanf/%y`. This function belongs to `bav_symbol.c`. It is run by 289 tests, including 35 in the `bav` library and 55 in the `DifferentialAlgebra` package.

`void * bav_scanf_default_symbol (void *z)` [Function]
The default function for parsing symbols, called by `ba0_scanf/%y`. A symbol is denoted by any indexed string the last indexed string indices of which is not formed of derivations. Exception `BAV_ERRUSY` is raised if the symbol is not recognized. This function belongs to `bav_symbol.c`. It is run by 289 tests, including 35 in the `bav` library and 55 in the `DifferentialAlgebra` package.

`void * bav_scanf2_symbol (void *z)` [Function]
Variant of `bav_scanf_default_symbol` which requires the symbol to be followed by a sequence of indexed string indices. The sequence is not part of the symbol. Exception `BAV_ERRUSY` is raised if the symbol is not recognized. Exception `BA0_ERRSYN` if the mandatory sequence of indexed string indices is missing. This function belongs to `bav_symbol.c`. It is run by 105 tests, including 0 in the `bav` library and 11 in the `DifferentialAlgebra` package.

`void bav_printf_symbol (void *z)` [Function]
The general printing function for symbols. It is called by `ba0_printf/%y`. This function belongs to `bav_symbol.c`. It is run by 396 tests, including 69 in the `bav` library and 63 in the `DifferentialAlgebra` package.

`void bav_printf_default_symbol (void *z)` [Function]
The default function for printing symbols (the `ident` field is printed). It is called by `ba0_printf/%y`. This function belongs to `bav_symbol.c`. It is run by 396 tests, including 69 in the `bav` library and 63 in the `DifferentialAlgebra` package.

`void bav_printf_numbered_symbol (void *z)` [Function]
A possible function for printing symbols (an underscore followed by the value of the `index_in_syms` field of the symbol. This function belongs to `bav_symbol.c`. It is not run by any test.

`void bav_R_set_symbol (struct bav_symbol *dst, struct bav_symbol *src, char *ident)` [Function]
This low level function assigns *src* to *dst*. The field `ident` of *dst* is however given by *ident*. This function belongs to `bav_symbol.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_symbol * bav_R_string_to_existing_symbol (char *s)` [Function]
Convert *s* to an existing symbol. Return `BAV_NOT_A_SYMBOL` if the symbol does not exist. This function belongs to `bav_differential_ring.c`. It is run by 449 tests, including 94 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`struct bav_symbol * bav_R_string_to_existing_derivation (char *s)` [Function]
Convert *s* to an existing independent symbol. Return `BAV_NOT_A_SYMBOL` if the symbol does not exist. This function belongs to `bav_differential_ring.c`. It is run by 385 tests, including 86 in the `bav` library and 64 in the `DifferentialAlgebra` package.

5.10 Dictionaries with symbol keys

Description of type `bav_dictionary_symbol`. This data structure implements dictionaries which map `struct bav_symbol *` to other objects. In this implementation, the values associated to the keys are integers, which are supposed to be indices in some table of objects of unspecified type. This data structure actually is as an alias for `struct ba0_dictionary`. This type declaration belongs to `bav_dictionary_symbol.h`.

`struct bav_dictionary_symbol`

```
{
// each entry contains either -1 or an index in some table of symbols
  struct ba0_tableof_int_p area;
// the shift applied on keys before hashing them
  ba0_int_p shift;
// the base-2 logarithm of the size/alloc field of area if area is nonempty
  ba0_int_p log2_size;
// the number of used entries in area
  ba0_int_p used_entries;
};
```

`void bav_init_dictionary_symbol (struct bav_dictionary_symbol *D, ba0_int_p log2_size)` [Function]

Initialize D to the empty dictionary. The argument *log2_size* provides the initial size of the field `area` of D. It should be at least equal to 8. This function belongs to `bav_dictionary_symbol.c`. It is run by 185 tests, including 0 in the `bav` library and 30 in the `DifferentialAlgebra` package.

`void bav_reset_dictionary_symbol (struct bav_dictionary_symbol *D)` [Function]

Reset D to the empty dictionary. This function belongs to `bav_dictionary_symbol.c`. It is run by 99 tests, including 0 in the `bav` library and 10 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_dictionary_symbol (struct bav_dictionary_symbol *D, enum ba0_garbage_code code)` [Function]

Return the size needed to perform a copy of D. If *code* is `ba0_embedded` then D is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `bav_dictionary_symbol.c`. It is not run by any test.

`void bav_set_dictionary_symbol (struct bav_dictionary_symbol *dst, struct bav_dictionary_symbol *src)` [Function]

Assign *src* to *dst*. This function belongs to `bav_dictionary_symbol.c`. It is not run by any test.

`ba0_int_p bav_get_dictionary_symbol (struct bav_dictionary_symbol *D, struct bav_tableof_symbol *T, struct bav_symbol *y)` [Function]

Return the index in T of the element associated to y. The element must have been recorded in D. Return `BA0_NOT_AN_INDEX` if not found. This function belongs to `bav_dictionary_symbol.c`. It is run by 128 tests, including 0 in the `bav` library and 13 in the `DifferentialAlgebra` package.

`void bav_add_dictionary_symbol (struct bav_dictionary_symbol *D, struct bav_tableof_symbol *T, struct bav_symbol *y, ba0_int_p z)` [Function]

Add to D an entry for the zth element of T which has key y. The element may actually not yet be present in T. Exception `BA0_ERRKEY` is raised if y is already present in D. This function belongs to `bav_dictionary_symbol.c`. It is run by 128 tests, including 0 in the `bav` library and 13 in the `DifferentialAlgebra` package.

5.11 Parameters

Description of type `bav_parameters`. This data type is only used to form the field `pars` of the `bav_differential_ring` structure. It handles the parameters of the differential ring.

The field `pars` contains the table of all parameters. Each range indexed group occurring in any element of the table actually is a range indexed string i.e. has a single radical.

The field `dict` maps strings to non-plain parameters in `pars`. The keys are the radicals of the range indexed strings. The dictionary does not contain any entry corresponding to plain parameters. This type declaration belongs to `bav_parameters.h`.

```
struct bav_parameters
{
// maps strings to radicals of range indexed strings for non-plain parameters
  struct ba0_dictionary_string dict;
// the table of parameters - range indexed groups are range indexed strings
  struct bav_tableof_parameter pars;
};
```

Description of type `bav_parameter`. This data structure permits to represent a *parameter* i.e. a dependent symbol / order zero dependent variable for which some derivatives are supposed to simplify to zero (the notion of parameter is here more general than the usual one).

A parameter may be a *plain parameter* or a parameter which fits a range indexed group. If the parameter is a plain one, then its identifier is the identifier of a dependent symbol / order zero dependent variable. If it fits a range indexed group, it describes a set of parameters which is made of all the symbols which fit the group.

The field `dependencies` contains the derivation indices of the derivations / independent variables the parameter depends on. If this field is empty, every derivative of the parameter is zero. The order of appearance of the derivations in this field is followed by some printing functions. This type declaration belongs to `bav_parameter.h`.

```
struct bav_parameter
{
// the range indexed group which describes the parameter identifiers
  struct ba0_range_indexed_group rig;
// the table of the derivation indices the parameter depends on
  struct ba0_tableof_int_p dependencies;
};
```

Description of type `bav_tableof_parameter`. This type declaration belongs to `bav_parameter.h`.

```
struct bav_tableof_parameter
{
  ba0_int_p alloc;
  ba0_int_p size;
  struct bav_parameter **tab;
};
```

`void bav_set_settings_parameter (char *`*`Function_PFE`*`)` [Function]

Set to *Function_PFE* the corresponding settings variable. This function belongs to `bav_parameter.c`. It is run by 65 tests, including 0 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_get_settings_parameter (char * *Function_PFE)` [Function]
Assign to **Function_PFE* the corresponding settings variable. All arguments are allowed to be zero. This function belongs to `bav_parameter.c`. It is run by 2 tests, including 0 in the `bav` library and 2 in the `DifferentialAlgebra` package.

`void bav_init_parameters (struct bav_parameters *pars)` [Function]
Initialize *pars*. This function belongs to `bav_parameters.c`. It is run by 494 tests, including 102 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_parameters (struct bav_parameters *pars, enum ba0_garbage_code code, bool strings_not_copied)` [Function]
Return the size needed to copy *pars*. If *code* is `ba0_embedded` then *pars* is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* then the strings occurring in *b* are supposed not to be copied. This function belongs to `bav_parameters.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_parameters (struct bav_parameters *dst, struct bav_parameters *src, struct bav_differential_ring *R)` [Function]
Copy *src* into *dst* without performing any string allocation. The strings occurring in *src* are supposed to be present in *R*. Each time a string has to be assigned, its copy in *R* is used. The size needed for the copy is the one returned by `bav_sizeof_parameters`. This function belongs to `bav_parameters.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_parameters_tableof_parameter (struct bav_parameters *pars, struct bav_tableof_parameter *T)` [Function]
Fill *pars* with the content of *T*. This function is called by the parser of orderings, in order to fill the `pars` field of `bav_global.R`. This function belongs to `bav_parameters.c`. It is run by 490 tests, including 101 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_init_parameter (struct bav_parameter *p)` [Function]
Initialize *p*. This function belongs to `bav_parameter.c`. It is run by 494 tests, including 102 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_parameter * bav_new_parameter (void)` [Function]
Allocate a new parameter, initialize it and return it. This function belongs to `bav_parameter.c`. It is run by 247 tests, including 38 in the `bav` library and 28 in the `DifferentialAlgebra` package.

`void bav_set_parameter (struct bav_parameter *dst, struct bav_parameter *src)` [Function]
Assign *src* to *dst*. This function belongs to `bav_parameter.c`. It is run by 247 tests, including 38 in the `bav` library and 28 in the `DifferentialAlgebra` package.

`void bav_set_parameter_with_tableof_string (struct bav_parameter *dst, struct bav_parameter *src, struct ba0_dictionary_string *D, struct ba0_tableof_string *T)` [Function]

Assign *src* to *dst* without any string duplication. Indeed, each string occurring in *src* is supposed to have a copy in T, whose index is supposed to be registered in D. Instead of duplicating strings, the copies are used. Exception `BA0_ERRALG` is raised if some string is not found in D. This function belongs to `bav_parameter.c`. It is run by 246 tests, including 37 in the `bav` library and 28 in the `DifferentialAlgebra` package.

`void bav_set_tableof_parameter_with_tableof_string (struct bav_tableof_parameter *dst, struct bav_tableof_parameter *src, struct ba0_dictionary_string *D, struct ba0_tableof_string *T)` [Function]

Assign *src* to *dst* without any string duplication by calling `bav_set_parameter_with_tableof_string`. This function belongs to `bav_parameter.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_parameter (struct bav_parameter *p, enum ba0_garbage_code code, bool strings_not_copied)` [Function]

Return the size needed to perform a copy of p. If *code* is `ba0_embedded` then p is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* is `true` then the strings occurring in p are supposed not to be copied. Return the size needed to perform a copy of T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. If *strings_not_copied* is `true` then the strings occurring in T are supposed not to be copied. This function belongs to `bav_parameter.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_tableof_parameter (struct bav_tableof_parameter *T, enum ba0_garbage_code code, bool strings_not_copied)` [Function]

This function belongs to `bav_parameter.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`bool bav_is_a_parameter (struct bav_symbol *y, struct bav_parameter **p)` [Function]

Return `true` if y is a parameter, else `false`. In the first case and if p is nonzero then $*p$ is assigned the address of the `struct bav_parameter` structure associated to y. This function belongs to `bav_parameter.c`. It is run by 342 tests, including 21 in the `bav` library and 61 in the `DifferentialAlgebra` package.

`void bav_annihilating_derivations_of_parameter (struct bav_tableof_symbol *T, struct bav_parameter *p)` [Function]

Assign to T all the derivations x such that the derivative of p with respect to x is zero. This function belongs to `bav_parameter.c`. It is run by 66 tests, including 0 in the `bav` library and 11 in the `DifferentialAlgebra` package.

`bool bav_is_zero_derivative_of_parameter (struct bav_variable *v)` [Function]

Return `true` if v is a dependent variable and a derivative of some parameter which is supposed to simplify to zero. This function belongs to `bav_parameter.c`. It is run by 269 tests, including 12 in the `bav` library and 53 in the `DifferentialAlgebra` package.

`void * bav_scanf_parameter (void *z)` [Function]

The general parsing function for parameters. It can be called via `ba0_scanf/%param`. A parameter is either a symbol or a symbol followed by a sequence of derivations, enclosed by parentheses. The sequence of derivations provides the dependencies of the parameter. This function belongs to `bav_parameter.c`. It is run by 247 tests, including 38 in the `bav` library and 28 in the `DifferentialAlgebra` package.

`void bav_printf_parameter_dependencies (struct bav_parameter *p)` [Function]

This subfunction of `bav_printf_parameter` prints the dependencies of p. This function belongs to `bav_parameter.c`. It is run by 17 tests, including 6 in the `bav` library and 10 in the `DifferentialAlgebra` package.

`void bav_printf_parameter (void *z)` [Function]

General printing function for parameters. It can be called via `ba0_printf/%param`. This function belongs to `bav_parameter.c`. It is run by 17 tests, including 6 in the `bav` library and 10 in the `DifferentialAlgebra` package.

5.12 Variables

Description of type `bav_variable`. A *variable* is a basic objects which permits to form terms and polynomials. A variable which is not a proper derivative of some differential indeterminate is said to be a *radical variable* (this includes independent variables, order zero derivatives). Variables are allocated in `ba0_global.stack.quiet` and are stored in the array `bav_global.R.vars`.

The field `number` contains the number of the variable with respect to each ordering occurring in `bav_global.R.ords`. The greater the number, the greater the variables. Numbers change when new variables are created (by differentiation of existing variables).

The fields `order` and `derivative` are only meaningful for derivatives of differential indeterminates. The sizes of these tables are equal to the number of derivations. The ith entry of each of these tables is associated to the ith derivation i.e. the ith element of the table `bav_global.R.ders`. This type declaration belongs to `bav_variable.h`.

```
struct bav_variable
{
// the symbol at the root of the variable
  struct bav_symbol *root;
// the index of the variable in bav_global.R.vars
  ba0_int_p index_in_vars;
// the size of number is equal to bav_global.R.ords.size
  struct ba0_tableof_int_p number;
// the two next fields are for derivatives of differential indeterminates
// or differential operators. Their size is equal to bav_global.R.ders.size
  struct ba0_tableof_int_p order;
```

```
  struct bav_tableof_variable derivative;
};
```

`struct bav_variable * bav_new_variable (void)` [Function]
: Allocate a new variable and return its address This function belongs to `bav_variable.c`. It is run by 487 tests, including 98 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_not_a_variable (void)` [Function]
: Return the constant `BAV_NOT_A_VARIABLE`. This function belongs to `bav_variable.c`. It is run by 467 tests, including 96 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_symbol_to_variable (struct bav_symbol *y)` [Function]
: Convert the symbol y to a variable. This function belongs to `bav_differential_ring.c`. It is run by 259 tests, including 8 in the `bav` library and 60 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_smallest_greater_variable (struct bav_variable *v)` [Function]
: Return the smallest variable strictly greater than v with respect to the current ordering. Return `BAV_NOT_A_VARIABLE` if it does not exist. This function belongs to `bav_differential_ring.c`. It is run by 39 tests, including 0 in the `bav` library and 9 in the `DifferentialAlgebra` package.

`ba0_int_p bav_block_index_symbol (struct bav_symbol *y)` [Function]
: Return the block index y belongs to in the table of blocks of the current ordering. This function belongs to `bav_ordering.c`. It is run by 29 tests, including 0 in the `bav` library and 6 in the `DifferentialAlgebra` package.

`ba0_int_p bav_block_index_range_indexed_group (struct ba0_range_indexed_group *rig)` [Function]
: Return the lowest block index the range indexed strings of *rig* belong to in the table of blocks of the current ordering. Return `BA0_NOT_AN_INDEX` if one of the range indexed strings is not recognized. This function belongs to `bav_ordering.c`. It is run by 23 tests, including 0 in the `bav` library and 6 in the `DifferentialAlgebra` package.

`ba0_int_p bav_block_index_parameter (struct bav_parameter *p)` [Function]
: Return the block index p belongs to in the table of blocks of the current ordering. Return `BA0_NOT_AN_INDEX` if the parameter is not recognized. This function belongs to `bav_ordering.c`. It is not run by any test.

`struct bav_variable * bav_derivation_index_to_derivation (ba0_int_p k)` [Function]
: Return the variable corresponding to the derivation index k i.e. the variable which has index k in `bav_global.R.ders`. This function belongs to `bav_differential_ring.c`. It is run by 318 tests, including 56 in the `bav` library and 60 in the `DifferentialAlgebra` package.

`void bav_set_settings_variable (ba0_scanf_function *s, ba0_printf_function *p, char *jet0_input_string, char *jet0_output_string, char *temp)` [Function]

Assign s and p to the corresponding setting variables. Set the input and output strings of the `jet0` notation. Set the prefix used for temporary variables to *tmp*. Zero values are replaced by default values. This function belongs to `bav_variable.c`. It is run by 370 tests, including 25 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_get_settings_variable (ba0_scanf_function **s, ba0_printf_function **p, char **jet0_input_string, char **jet0_output_string, char **temp)` [Function]

Assign to s and p the values of the corresponding setting variable. Assign to *jet0_input_string* and *jet0_output_string* the corresponding strings. Assign to *tmp* the prefix used for temporary variables. Parameters may be zero. This function belongs to `bav_variable.c`. It is run by 65 tests, including 0 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_switch_ring_variable (struct bav_variable *v, struct bav_differential_ring *R)` [Function]

Return the variable of R which has the same index in `R->vars` as v. This low level function should be used in conjunction with `bav_set_differential_ring`: if R is a ring obtained by application of `bav_set_differential_ring` to the ring v refers to, then this function returns the element of R which corresponds to v. This function belongs to `bav_variable.c`. It is run by 38 tests, including 0 in the `bav` library and 36 in the `DifferentialAlgebra` package.

`bav_Iorder bav_order_variable (struct bav_variable *v, struct bav_symbol *d)` [Function]

Return the order of v with respect to d. If v is not a derivative or an operator or if d is not a derivation, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 103 tests, including 1 in the `bav` library and 18 in the `DifferentialAlgebra` package.

`bav_Iorder bav_total_order_variable (struct bav_variable *v)` [Function]

Return the total order of v. If v is not a derivative or an operator, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 459 tests, including 94 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`bool bav_is_constant_variable (struct bav_variable *v, struct bav_symbol *s)` [Function]

Return `true` if v is a constant with respect to derivation s (or with respect to all derivations, if s is `BAV_NOT_A_SYMBOL`). This function belongs to `bav_parameter.c`. It is run by 63 tests, including 0 in the `bav` library and 10 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_diff_variable (struct bav_variable *v, struct bav_symbol *s)` [Function]

Return the derivative of v with respect to s. If v is not a derivative or an operator or if s is not a derivation, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 367 tests, including 70 in the `bav` library and 61 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_diff2_variable (struct bav_variable *v, struct bav_term *theta)` [Function]

Return the derivative of v with respect to the derivation operator encoded by *theta*. If v is not a derivative or an operator or if some factor of *theta* is not a derivation, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 61 tests, including 0 in the `bav` library and 11 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_int_variable (struct bav_variable *v, struct bav_symbol *s)` [Function]

Return the variable u whose derivative with respect to s is equal to v. If v is not a derivative or an operator, if s is not a derivation, or v has zero order with respect to s, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 14 tests, including 1 in the `bav` library and 3 in the `DifferentialAlgebra` package.

`enum bav_typeof_symbol bav_symbol_type_variable (struct bav_variable *v)` [Function]

Return the type of the symbol over which v is built. This function belongs to `bav_variable.c`. It is run by 270 tests, including 6 in the `bav` library and 50 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_order_zero_variable (struct bav_variable *v)` [Function]

Return the order zero variable v is a derivative of. If v is not a derivative or an operator, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 43 tests, including 1 in the `bav` library and 9 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_lcd_variable (struct bav_variable *v, struct bav_variable *w)` [Function]

Return the least common derivative of v and w. If v and w are not both differential operators, or if they are not derivatives of the same differential indeterminate, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 131 tests, including 0 in the `bav` library and 26 in the `DifferentialAlgebra` package.

`bool bav_disjoint_variables (struct bav_variable *v, struct bav_variable *w)` [Function]

Return `true` if, for every derivation s, the order of v with respect to s is zero or the order of w with respect to s is zero. If v and w are not derivatives of the same differential indeterminate, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is not run by any test.

`bool bav_is_derivative (struct bav_variable *v, struct bav_variable *w)` [Function]

Return `true` if v is a derivative of w else `false`. This function belongs to `bav_variable.c`. It is run by 229 tests, including 2 in the `bav` library and 35 in the `DifferentialAlgebra` package.

`bool bav_is_proper_derivative (struct bav_variable *v, struct bav_variable *w)` [Function]

Return `true` if v is a proper derivative of w else `false`. This function belongs to `bav_variable.c`. It is run by 135 tests, including 0 in the `bav` library and 28 in the `DifferentialAlgebra` package.

`bool bav_is_d_derivative (struct bav_variable *v, struct bav_variable *w, struct bav_symbol *d)` [Function]

Return `true` if v is the derivative of w with respect to derivation d else `false`. Exception `BAO_ERRALG` is raised if d is not an independent variable. This function belongs to `bav_variable.c`. It is run by 121 tests, including 38 in the `bav` library and 25 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_derivation_between_derivatives (struct bav_variable *v, struct bav_variable *w)` [Function]

Return a derivation d such that v (resp. w) is a derivative of the derivative of w (resp. v) with respect to d. If v and w are not derivatives of the same differential indeterminate, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 168 tests, including 1 in the `bav` library and 32 in the `DifferentialAlgebra` package.

`void bav_operator_between_derivatives (struct bav_term *T, struct bav_variable *v, struct bav_variable *w)` [Function]

Assign to T the power product of derivations such that v (resp. w) is the derivative w (resp. v) with respect to T. If v and w are not derivatives of the same differential indeterminate, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 110 tests, including 1 in the `bav` library and 26 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_next_derivative (struct bav_variable *v, struct bav_tableof_variable *T)` [Function]

This function permits to enumerate the derivatives by increasing total order. It returns the derivative which follows v and which is not a derivative of any element of T. If no such derivative exists, it returns `BAV_NOT_A_VARIABLE`. If v and some element of T are not derivatives of the same differential indeterminate, exception `BAO_ERRALG` is raised. This function belongs to `bav_variable.c`. It is run by 1 tests, including 1 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_indexed_string_to_variable (struct ba0_indexed_string *indexed)` [Function]

Return the variable identified by *indexed*. The variable may already exist but may also need to be created, if *indexed* fits some range indexed group in `bav_global.R.rigs`. Note that the function does not handle trailing indexed string indices forming a

derivation operator in the jet notation: if a dependent variable is returned it has order zero. This function belongs to `bav_variable.c`. It is run by 485 tests, including 97 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_variable *` [Function]
`bav_index_in_rigs_and_subscript_to_variable (ba0_int_p` *`index_in_rigs`*`, ba0_int_p` *`subscript`*`)`

Return the order zero variable *ident[subscript]* where *ident* is the radical of the range indexed string defined by *index_in_rigs*. Return `BAV_NOT_A_VARIABLE` if *subscript* does not fit the indices of *ident*. This function belongs to `bav_variable.c`. It is run by 4 tests, including 1 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void * bav_scanf_variable (void *`*`z`*`)` [Function]

This function belongs to `bav_variable.c`. It is run by 480 tests, including 96 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`void * bav_scanf_jet_variable (void *`*`z`*`)` [Function]

The function for parsing variables in the `jet` notation. Empty square brackets are allowed but not mandatory behind order zero derivatives. The default function called by `ba0_scanf/%v`. This function belongs to `bav_variable.c`. It is run by 141 tests, including 96 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void * bav_scanf_jet0_variable (void *`*`z`*`)` [Function]

The function for parsing variables in the `jet0` notation. Order zero variables are supposed to be indexed by `[0]`. However, empty square brackets and no square brackets at all are allowed. The `0` character can be customized by modifying `bav_initialized_global.variable.jet0_input_string`. A function that may be called by `ba0_scanf/%v`. This function belongs to `bav_variable.c`. It is not run by any test.

`void * bav_scanf_diff_variable (void *`*`z`*`)` [Function]

The function for parsing variables either in the Maple `diff` or `Diff` notations. This function may be called by `ba0_scanf/%v`. This function belongs to `bav_variable.c`. It is run by 3 tests, including 1 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void * bav_scanf_inert_diff_variable (void *`*`z`*`)` [Function]

This function belongs to `bav_variable.c`. It is not run by any test.

`void * bav_scanf_python_Derivative_variable (void *`*`z`*`)` [Function]

The function for parsing variables in the Python/sympy `Derivative` notation. This function may be called by `ba0_scanf/%v`. This function belongs to `bav_variable.c`. It is run by 39 tests, including 1 in the `bav` library and 38 in the `DifferentialAlgebra` package.

`void * bav_scanf_python_D_variable (void *`*`z`*`)` [Function]

The function for parsing variables in the Sagemath `D` notation. A function that may be called by `ba0_scanf/%v`. This function belongs to `bav_variable.c`. It is run by 3 tests, including 2 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void * bav_scanf_maple_D_variable (void *z)` [Function]
The function for parsing variables in the Maple `D` notation. A function that may be called by `ba0_scanf/%v`. This function belongs to `bav_variable.c`. It is not run by any test.

`void * bav_scanf_python_all_variable (void *z)` [Function]
The function used for parsing variables from Python and Sagemath. The set of compatible notations is stored in the global variable `bav_global.variable.notations`. It is called by `ba0_scanf/%v`. This function belongs to `bav_variable.c`. It is run by 64 tests, including 0 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`void * bav_scanf_maple_all_variable (void *z)` [Function]
The function for parsing variables in Maple. The set of compatible notations is stored in the global variable `bav_global.variable.notations`. A function that may be called by `ba0_scanf/%v`. This function belongs to `bav_variable.c`. It is run by 287 tests, including 14 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void bav_reset_notations (void)` [Function]
Set `bav_global.variable.notations` to all notations. This function belongs to `bav_variable.c`. It is not run by any test.

`ba0_int_p bav_get_notations (void)` [Function]
Return `bav_global.variable.notations`. This function belongs to `bav_variable.c`. It is not run by any test.

`void bav_printf_variable (void *z)` [Function]
This function belongs to `bav_variable.c`. It is run by 396 tests, including 69 in the `bav` library and 63 in the `DifferentialAlgebra` package.

`void bav_printf_jet_variable (void *z)` [Function]
Print a variable following in the `jet` notation. The default function called by `ba0_printf/%v`. This function belongs to `bav_variable.c`. It is run by 344 tests, including 60 in the `bav` library and 21 in the `DifferentialAlgebra` package.

`void bav_printf_jet_wesb_variable (void *z)` [Function]
The function for printing variables in the `tjet` notation. Order zero derivatives are followed by empty square brackets. This function can be called by `ba0_printf/%v`. This function belongs to `bav_variable.c`. It is not run by any test.

`void bav_printf_jet0_variable (void *z)` [Function]
The function for printing variables in the `jet0` notation. Order zero derivatives are followed by `[0]`. The `0` character can be customized by modifying `bav_initialized_global.variable.jet0_output_string` This function can be called by `ba0_printf/%v`. This function belongs to `bav_variable.c`. It is run by 3 tests, including 0 in the `bav` library and 3 in the `DifferentialAlgebra` package.

`void bav_printf_LaTeX_variable (void *z)` [Function]
The function for printing variables in the `jet` notation using LaTeX indices. This function can be called by `ba0_printf/%v`. This function belongs to `bav_variable.c`. It is not run by any test.

`void bav_printf_maple_D_variable (void *z)` [Function]
The function for printing variables in the Maple `D` notation. This function can be called by `ba0_printf/%v`. This function belongs to `bav_variable.c`. It is run by 1 tests, including 1 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void bav_printf_diff_variable (void *z)` [Function]
The function for printing variables in the Maple `diff` notation. This function can be called by means of `ba0_printf/%v`. This function belongs to `bav_variable.c`. It is run by 49 tests, including 8 in the `bav` library and 39 in the `DifferentialAlgebra` package.

`void bav_printf_inert_diff_variable (void *z)` [Function]
The function for printing variables in the Maple `Diff` notation. This function can be called by `ba0_printf/%v`. This function belongs to `bav_variable.c`. It is not run by any test.

`void bav_printf_python_Derivative_variable (void *z)` [Function]
The function for printing variables in the Python/sympy `Derivative` notation. This function can be called by `ba0_printf/%v`. This function belongs to `bav_variable.c`. It is run by 42 tests, including 3 in the `bav` library and 39 in the `DifferentialAlgebra` package.

`bool bav_gt_index_variable (void *x, void *y)` [Function]
Return `true` if the field `index_in_vars` of x is greater than that of y. This function belongs to `bav_variable.c`. It is not run by any test.

`bool bav_gt_variable (void *x, void *y)` [Function]
Return `true` if x is greater than y with respect to the current ordering. This function belongs to `bav_variable.c`. It is run by 169 tests, including 33 in the `bav` library and 28 in the `DifferentialAlgebra` package.

`void bav_sort_tableof_variable (struct bav_tableof_variable *T, enum ba0_sort_mode mode)` [Function]
Sort T in place in ascending order with respect to the current ordering, if *mode* is `ba0_ascending_mode`, or in descending order, if *mode* is `ba0_descending_mode`. This function belongs to `bav_variable.c`. It is run by 157 tests, including 35 in the `bav` library and 26 in the `DifferentialAlgebra` package.

`void bav_block_sort_tableof_variable (struct bav_tableof_tableof_variable *T, struct bav_tableof_variable *R, struct bav_tableof_variable *X)` [Function]
Assign each variable present in X to a table of T or to R. Dependent variables which are moreover not present in the `varmax` or `varmin` fields of the current ordering are assigned to a table of T. The other variables are assigned to R. Each entry of T is assigned variables which belong to the same block of the current ordering. Leftmost blocks are higher than rightmost ones. The entries of T are nonempty and are sorted in decreasing order. This function belongs to `bav_ordering.c`. It is run by 6 tests, including 6 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void bav_independent_variables (struct bav_tableof_variable *T)` [Function]

Assign to T all the derivations / independent variables. This function belongs to `bav_variable.c`. It is not run by any test.

`bav_Iordering bav_variable_number (struct bav_variable *v)` [Function]

Return the number of v with respect to the current ordering. This function belongs to `bav_differential_ring.c`. It is run by 417 tests, including 55 in the `bav` library and 58 in the `DifferentialAlgebra` package.

`void bav_R_set_variable (struct bav_variable *dst, struct bav_variable *src, struct bav_differential_ring *R)` [Function]

This low level function assigns *src* to *dst*. All the pointer fields of *src* which point to the differential ring structure *src* belongs to are updated to point towards the corresponding symbols and variables of R. This function belongs to `bav_variable.c`. It is run by 74 tests, including 7 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_R_new_derivative (struct bav_variable *v, struct bav_symbol *d)` [Function]

Return the derivative of v with respect to d assuming this derivative is not already stored in the `derivative` field of v. This low level function is called by `bav_diff_variable`. This function belongs to `bav_differential_ring.c`. It is run by 367 tests, including 70 in the `bav` library and 61 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_R_string_to_existing_variable (char *s)` [Function]

Convert s to an existing variable. Return `BAV_NOT_A_VARIABLE` if the variable does not exist. This function belongs to `bav_differential_ring.c`. It is run by 485 tests, including 97 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_R_set_maximal_variable (struct bav_variable *v)` [Function]

Change the current ordering in such a way that v becomes the greatest variable (insert v at the beginning of the field `varmax` of the current ordering). The ordering between other variables is left unchanged. This function belongs to `bav_differential_ring.c`. It is run by 298 tests, including 23 in the `bav` library and 41 in the `DifferentialAlgebra` package.

`void bav_R_set_minimal_variable (struct bav_variable *v)` [Function]

Change the current ordering in such a way that v becomes the lowest variable (insert v at the end of the field `varmin` of the current ordering). The ordering between other variables is left unchanged. This function belongs to `bav_differential_ring.c`. It is run by 48 tests, including 9 in the `bav` library and 9 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_R_new_temporary_variable (void)` [Function]

Return a new temporary variable. Its symbol has type `bav_temporary_symbol`. This function belongs to `bav_differential_ring.c`. It is run by 1 tests, including 1 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void bav_R_free_temporary_variable (struct bav_variable *v)` [Function]
Declare the temporary variable *v* as logically free. The next call to `bav_R_new_temporary_variable` may then return *v* instead of creating some new symbol. This function belongs to `bav_differential_ring.c`. It is run by 1 tests, including 1 in the `bav` library and 0 in the `DifferentialAlgebra` package.

5.13 Dictionaries with variable keys

Description of type `bav_dictionary_variable`. This data structure implements dictionaries which map `struct bav_variable *` to other objects. In this implementation, the values associated to the keys are integers, which are supposed to be indices in some table of objects of unspecified type. This data structure actually is as an alias for `struct ba0_dictionary`. This type declaration belongs to `bav_dictionary_variable.h`.

```
struct bav_dictionary_variable
{
// each entry contains either -1 or an index in some table of variables
  struct ba0_tableof_int_p area;
// the shift applied on keys before hashing them
  ba0_int_p shift;
// the base-2 logarithm of the size/alloc field of area if area is nonempty
  ba0_int_p log2_size;
// the number of used entries in area
  ba0_int_p used_entries;
};
```

`void bav_init_dictionary_variable (struct bav_dictionary_variable *D, ba0_int_p log2_size)` [Function]
Initialize *D* to the empty dictionary. The argument *log2_size* provides the initial size of the field `area` of *D*. It should be at least equal to 8. This function belongs to `bav_dictionary_variable.c`. It is run by 384 tests, including 3 in the `bav` library and 63 in the `DifferentialAlgebra` package.

`void bav_reset_dictionary_variable (struct bav_dictionary_variable *D)` [Function]
Reset *D* to the empty dictionary. This function belongs to `bav_dictionary_variable.c`. It is run by 329 tests, including 0 in the `bav` library and 53 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bav_sizeof_dictionary_variable (struct bav_dictionary_variable *D, enum ba0_garbage_code code)` [Function]
Return the size needed to perform a copy of *D*. If *code* is `ba0_embedded` then *D* is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `bav_dictionary_variable.c`. It is not run by any test.

`void bav_set_dictionary_variable (struct bav_dictionary_variable *dst, struct bav_dictionary_variable *src)` [Function]
Assign *src* to *dst*. This function belongs to `bav_dictionary_variable.c`. It is not run by any test.

`ba0_int_p bav_get_dictionary_variable (struct bav_dictionary_variable *D, struct bav_tableof_variable *T, struct bav_variable *v)` [Function]

Return the index in T of the element associated to v. The element must have been recorded in D. Return `BA0_NOT_AN_INDEX` if not found. This function belongs to `bav_dictionary_variable.c`. It is run by 375 tests, including 3 in the `bav` library and 60 in the `DifferentialAlgebra` package.

`void bav_add_dictionary_variable (struct bav_dictionary_variable *D, struct bav_tableof_variable *T, struct bav_variable *v, ba0_int_p z)` [Function]

Add to D an entry for the zth element of T which has key v. The element may actually not yet be present in T. Exception `BA0_ERRKEY` is raised if v is already present in D. This function belongs to `bav_dictionary_variable.c`. It is run by 375 tests, including 3 in the `bav` library and 60 in the `DifferentialAlgebra` package.

5.14 Ranks

Description of type `bav_rank`. A *rank* is a variable raised to some degree. Some special ranks are also defined such as the rank of zero and the one of nonzero constants. The rank of zero is encoded by (`BAV_NOT_A_VARIABLE`, -1). The rank of nonzero constants is encoded by a field `deg` equal to 0 (the field `var` is unspecified). This type declaration belongs to `bav_rank.h`.

```
struct bav_rank
{
// possibly BAV_NOT_A_VARIABLE
  struct bav_variable *var;
// possibly zero or a negative integer
  bav_Idegree deg;
};
```

`void bav_set_settings_rank (ba0_printf_function *p)` [Function]

Customize to p the way ranks are printed. This function belongs to `bav_rank.c`. It is run by 341 tests, including 14 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_get_settings_rank (ba0_printf_function **p)` [Function]

Assign to p the function used for printing ranks. This function belongs to `bav_rank.c`. It is run by 37 tests, including 0 in the `bav` library and 7 in the `DifferentialAlgebra` package.

`void bav_init_rank (struct bav_rank *rg)` [Function]

Dummy constructor. This function belongs to `bav_rank.c`. It is run by 4 tests, including 1 in the `bav` library and 3 in the `DifferentialAlgebra` package.

`struct bav_rank * bav_new_rank (void)` [Function]

Allocates a new rank, initialize it and return it. This function belongs to `bav_rank.c`. It is run by 4 tests, including 1 in the `bav` library and 3 in the `DifferentialAlgebra` package.

`bool bav_is_zero_rank (struct bav_rank *rg)` [Function]
Return `true` if *rg* is the rank of zero. This function belongs to `bav_rank.c`. It is run by 402 tests, including 23 in the `bav` library and 63 in the `DifferentialAlgebra` package.

`bool bav_is_constant_rank (struct bav_rank *rg)` [Function]
Return `true` if *rg* is the rank of a nonzero constant. This function belongs to `bav_rank.c`. It is run by 402 tests, including 23 in the `bav` library and 63 in the `DifferentialAlgebra` package.

`bool bav_lt_rank (struct bav_rank *rg, struct bav_rank *sg)` [Function]
Return `true` if *rg* is less than *sg*. The rank of zero is lower than any other rank. The rank of nonzero constants is lower than classical ranks. Classical ranks are ordered by comparing first variables with respect to the current ordering and, in case of equality, the degrees. This function belongs to `bav_rank.c`. It is run by 249 tests, including 0 in the `bav` library and 38 in the `DifferentialAlgebra` package.

`bool bav_gt_rank (struct bav_rank *rg, struct bav_rank *sg)` [Function]
Return `true` if *rg* is greater than *sg*. This function belongs to `bav_rank.c`. It is run by 1 tests, including 0 in the `bav` library and 1 in the `DifferentialAlgebra` package.

`bool bav_equal_rank (struct bav_rank *rg, struct bav_rank *sg)` [Function]
Return `true` if *rg* and *sg* are equal. This function belongs to `bav_rank.c`. It is run by 116 tests, including 0 in the `bav` library and 30 in the `DifferentialAlgebra` package.

`int bav_compare_rank (const void *x, const void *y)` [Function]
Comparison function for `qsort`. When used by `qsort`, the resulting table of ranks is sorted by decreasing order with respect to the current ordering. This function is used to sort terms. This function belongs to `bav_rank.c`. It is run by 336 tests, including 4 in the `bav` library and 56 in the `DifferentialAlgebra` package.

`struct bav_rank bav_zero_rank (void)` [Function]
Return the rank of zero. This function belongs to `bav_rank.c`. It is run by 27 tests, including 1 in the `bav` library and 14 in the `DifferentialAlgebra` package.

`struct bav_rank bav_constant_rank (void)` [Function]
Return the rank of nonzero constants. This function belongs to `bav_rank.c`. It is run by 366 tests, including 1 in the `bav` library and 55 in the `DifferentialAlgebra` package.

`struct bav_rank bav_constant_rank2 (struct bav_variable *v)` [Function]
Return the rank of nonzero constant also (with *v* encoded in the field `var` of the result). This function belongs to `bav_rank.c`. It is run by 2 tests, including 0 in the `bav` library and 1 in the `DifferentialAlgebra` package.

`void bav_printf_default_rank (void *z)` [Function]
The default function for printing ranks. It can be customized by modifying `bav_initialized_global.rank.printf`. It is called by `ba0_printf/%rank`. Exponentiation is denoted using `^`. This function belongs to `bav_rank.c`. It is run by 70 tests, including 20 in the `bav` library and 0 in the `DifferentialAlgebra` package.

`void * bav_scanf_rank (void *z)` [Function]
The function called for parsing ranks. It is called by `ba0_scanf/%rank`. The character `0` denotes the rank of zero. Any positive integer denotes the rank of nonzero constants. Exponentiation can be denoted `^` or `**`. A variable to a zero exponent yields the rank of nonzero constants. Exponents may be negative. This function belongs to `bav_rank.c`. It is run by 7 tests, including 1 in the `bav` library and 2 in the `DifferentialAlgebra` package.

`void bav_printf_rank (void *z)` [Function]
This function belongs to `bav_rank.c`. It is run by 332 tests, including 20 in the `bav` library and 60 in the `DifferentialAlgebra` package.

`void bav_printf_stars_rank (void *z)` [Function]
A function for printing ranks. Exponentiation is denoted using `**`. This function belongs to `bav_rank.c`. It is run by 262 tests, including 0 in the `bav` library and 60 in the `DifferentialAlgebra` package.

`void bav_printf_list_rank (void *z)` [Function]
A function for printing ranks. Ranks are printed as a list `[variable, degree]`. This function belongs to `bav_rank.c`. It is run by 2 tests, including 0 in the `bav` library and 2 in the `DifferentialAlgebra` package.

5.15 Terms

Description of type `bav_term`. A *term* is a product of ranks (the ranks of zero and of nonzero constants are forbidden) sorted by decreasing order with respect to the current ordering. The empty product encodes the term 1. The leading rank of a nonempty product is the leading rank of the term. This type declaration belongs to `bav_term.h`.

```
struct bav_term
{
// number of entries allocated to rg
  ba0_int_p alloc;
// number of entries used in rg
  ba0_int_p size;
// the array of ranks, by decreasing order with respect to the current ordering
  struct bav_rank *rg;
};
```

`void bav_init_term (struct bav_term *T)` [Function]
Initialize T to the term 1. This function belongs to `bav_term.c`. It is run by 407 tests, including 22 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`struct bav_term * bav_new_term (void)` [Function]
Allocate a new term, initialize it and return it. This function belongs to `bav_term.c`. It is run by 350 tests, including 14 in the `bav` library and 58 in the `DifferentialAlgebra` package.

`void bav_switch_ring_term (struct bav_term *T, struct bav_differential_ring *R)` [Function]
Replace each variable occurring in T by the one obtained by applying `bav_switch_ring_variable` over it. See this function. The term T is modified. This low level

function should be used in conjunction with `bav_set_differential_ring`. This function belongs to `bav_term.c`. It is run by 38 tests, including 0 in the `bav` library and 36 in the `DifferentialAlgebra` package.

`void bav_realloc_term (struct bav_term *T, ba0_int_p n)` [Function]
Realloc the field `rg` if needed so that it can receive at least n ranks. The existing ranks are kept. This function belongs to `bav_term.c`. It is run by 407 tests, including 22 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`void bav_set_term_one (struct bav_term *T)` [Function]
Assign the term 1 to T. This function belongs to `bav_term.c`. It is run by 389 tests, including 6 in the `bav` library and 63 in the `DifferentialAlgebra` package.

`void bav_set_term_rank (struct bav_term *T, struct bav_rank *rg)` [Function]
Assign rg to T. If rg is the rank of nonzero constant then assigm the term 1 to T. Exception `BAV_ERRRGZ` is raised if rg is the rank of zero. This function belongs to `bav_term.c`. It is run by 365 tests, including 0 in the `bav` library and 55 in the `DifferentialAlgebra` package.

`void bav_set_term_variable (struct bav_term *T, struct bav_variable *v, bav_Idegree d)` [Function]
Assign v^d to T. If d is zero then assign the term 1 to T. This function belongs to `bav_term.c`. It is run by 381 tests, including 1 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`void bav_set_term (struct bav_term *T, struct bav_term *U)` [Function]
Assign U to T. This function belongs to `bav_term.c`. It is run by 494 tests, including 102 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`void bav_set_tableof_term (struct bav_tableof_term *T, struct bav_tableof_term *U)` [Function]
Assign U to T. This function belongs to `bav_term.c`. It is run by 59 tests, including 0 in the `bav` library and 10 in the `DifferentialAlgebra` package.

`void bav_shift_term (struct bav_term *T, struct bav_term *U)` [Function]
Assign to T the term obtained by removing the leading rank of U. May raise exception `BAV_ERRTEU` (unexpected term one). This function belongs to `bav_term.c`. It is run by 230 tests, including 0 in the `bav` library and 35 in the `DifferentialAlgebra` package.

`void bav_strip_term (struct bav_term *T, struct bav_term *U, bav_Inumber n)` [Function]
Assign to T the term obtained by removing from U all the variables with numbers strictly less than n. This function belongs to `bav_term.c`. It is run by 253 tests, including 0 in the `bav` library and 39 in the `DifferentialAlgebra` package.

`bool bav_is_one_term (struct bav_term *T)` [Function]
Return `true` if T is the term 1. This function belongs to `bav_term.c`. It is run by 384 tests, including 0 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`struct bav_variable * bav_leader_term (struct bav_term *T)` [Function]
Return the leading variable of T. May raise exception `BAV_ERRTEU` (unexpected term one). This function belongs to `bav_term.c`. It is run by 308 tests, including 0 in the `bav` library and 48 in the `DifferentialAlgebra` package.

`bav_Idegree bav_leading_degree_term (struct bav_term *T)` [Function]
Return the degree of the leading rank. May raise exception `BAV_ERRTEU` (unexpected term one). This function belongs to `bav_term.c`. It is run by 267 tests, including 0 in the `bav` library and 38 in the `DifferentialAlgebra` package.

`bav_Idegree bav_total_degree_term (struct bav_term *T)` [Function]
Return the total degree of the term: the sum of the degrees of all ranks. This function belongs to `bav_term.c`. It is run by 109 tests, including 1 in the `bav` library and 25 in the `DifferentialAlgebra` package.

`bav_Idegree bav_maximal_degree_term (struct bav_term *T)` [Function]
Return the maximum of the degrees of all ranks. This function belongs to `bav_term.c`. It is not run by any test.

`bav_Idegree bav_degree_term (struct bav_term *T, struct bav_variable *v)` [Function]
Return the degree of T with respect to v. This function belongs to `bav_term.c`. It is run by 128 tests, including 0 in the `bav` library and 26 in the `DifferentialAlgebra` package.

`bav_Iorder bav_total_order_term (struct bav_term *T)` [Function]
Return the maximum of the orders of the variables of T. This function belongs to `bav_term.c`. It is run by 1 tests, including 0 in the `bav` library and 1 in the `DifferentialAlgebra` package.

`struct bav_rank bav_leading_rank_term (struct bav_term *T)` [Function]
Return the leading rank of T. This function belongs to `bav_term.c`. It is run by 295 tests, including 0 in the `bav` library and 45 in the `DifferentialAlgebra` package.

`bool bav_equal_term (struct bav_term *T, struct bav_term *U)` [Function]
Return `true` if T and U are equal. This function belongs to `bav_term.c`. It is run by 383 tests, including 0 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`bool bav_gt_term (struct bav_term *T, struct bav_term *U)` [Function]
Return `true` if $T > U$ with respect to the lexicographic order induced by the current ordering. This function belongs to `bav_term.c`. It is run by 1 tests, including 0 in the `bav` library and 1 in the `DifferentialAlgebra` package.

`bool bav_lt_term (struct bav_term *T, struct bav_term *U)` [Function]
Return `true` if $T < U$ with respect to the lexicographic order induced by the current ordering. This function belongs to `bav_term.c`. It is run by 2 tests, including 0 in the `bav` library and 2 in the `DifferentialAlgebra` package.

`bool bav_disjoint_term (struct bav_term *T, struct bav_term *U)` [Function]

Return `true` if T and U have no variable in common i.e. if their lcm is equal to their product. This function belongs to `bav_term.c`. It is not run by any test.

`void bav_sort_term (struct bav_term *T)` [Function]

Sort T in place in decreasing order with respect to the current ordering. This function belongs to `bav_term.c`. It is run by 380 tests, including 4 in the `bav` library and 60 in the `DifferentialAlgebra` package.

`void bav_sort_tableof_term (struct bav_tableof_term *T)` [Function]

Apply `bav_sort_term` to each element of T. This function belongs to `bav_term.c`. It is not run by any test.

`void bav_lcm_term (struct bav_term *R, struct bav_term *T, struct bav_term *U)` [Function]

Assign to R the least common multiple of T and U. This function belongs to `bav_term.c`. It is run by 378 tests, including 0 in the `bav` library and 61 in the `DifferentialAlgebra` package.

`void bav_lcm_tableof_term (struct bav_tableof_term *R, struct bav_tableof_term *T, struct bav_tableof_term *U)` [Function]

Take pairwise least common multiples of T and U and assign them to R. The tables T and U may have different sizes. In such a case, the missing entries of the shortest table are supposed to be equal to 1. This function belongs to `bav_term.c`. It is run by 67 tests, including 0 in the `bav` library and 10 in the `DifferentialAlgebra` package.

`void bav_gcd_term (struct bav_term *R, struct bav_term *T, struct bav_term *U)` [Function]

Assign to R the greatest common divisor of T and U. This function belongs to `bav_term.c`. It is run by 328 tests, including 0 in the `bav` library and 53 in the `DifferentialAlgebra` package.

`void bav_mul_term (struct bav_term *R, struct bav_term *T, struct bav_term *U)` [Function]

Assign to R the product of T and U. This function belongs to `bav_term.c`. It is run by 373 tests, including 22 in the `bav` library and 58 in the `DifferentialAlgebra` package.

`void bav_mul_term_rank (struct bav_term *R, struct bav_term *T, struct bav_rank *rg)` [Function]

Assign to R the product of T by rg. Exception `BAV_ERRRGZ` is raised if rg is the rank of zero. This function belongs to `bav_term.c`. It is run by 271 tests, including 22 in the `bav` library and 43 in the `DifferentialAlgebra` package.

`void bav_mul_term_variable (struct bav_term *T, struct bav_term *U, struct bav_variable *v, bav_Idegree d)` [Function]

Assign to T the product of U by v^d. If d is zero then assign U to T. This function belongs to `bav_term.c`. It is run by 226 tests, including 1 in the `bav` library and 34 in the `DifferentialAlgebra` package.

`void bav_pow_term (struct bav_term *T, struct bav_term *U, bav_Idegree d)` [Function]

Assign to T the term U^d. If d is zero then assign the term 1 to T. This function belongs to `bav_term.c`. It is run by 212 tests, including 0 in the `bav` library and 33 in the `DifferentialAlgebra` package.

`bool bav_is_factor_term (struct bav_term *T, struct bav_term *U, struct bav_term *R)` [Function]

Return `true` if U is a factor of T. If so and R is a nonzero pointer then assign T/U to R. This function belongs to `bav_term.c`. It is run by 384 tests, including 0 in the `bav` library and 64 in the `DifferentialAlgebra` package.

`void bav_exquo_term (struct bav_term *R, struct bav_term *T, struct bav_term *U)` [Function]

Assign T/U to R. Raise the exception `BAV_EXEXQO` if U is not a factor of T. This function belongs to `bav_term.c`. It is run by 332 tests, including 0 in the `bav` library and 53 in the `DifferentialAlgebra` package.

`void bav_exquo_term_variable (struct bav_term *R, struct bav_term *T, struct bav_variable *x, bav_Idegree d)` [Function]

Assign T/x^d to R. Raise the exception `BAV_EXEXQO` if x^d is not a factor of T. This function belongs to `bav_term.c`. It is run by 110 tests, including 0 in the `bav` library and 11 in the `DifferentialAlgebra` package.

`void bav_diff_term (struct bav_term *R, struct bav_term *T, struct bav_symbol *x)` [Function]

Assign to R a multiple of all the terms of the differential polynomial obtained by differentiating T with respect to the independent variable x. This function belongs to `bav_term.c`. It is run by 196 tests, including 6 in the `bav` library and 42 in the `DifferentialAlgebra` package.

`void bav_set_term_tableof_variable (struct bav_term *T, struct bav_tableof_variable *vars, struct ba0_tableof_int_p *degs)` [Function]

Assign to T the product of the variables of *vars* raised to the degrees of *degs*. The two tables must have the same size. Zero degrees are allowed. This function belongs to `bav_term.c`. It is run by 378 tests, including 0 in the `bav` library and 61 in the `DifferentialAlgebra` package.

`bool bav_depends_on_zero_derivatives_of_parameter_term (struct bav_term *T)` [Function]

Return `true` if T depends on a derivative of a parameter which should simplify to zero. This function belongs to `bav_term.c`. It is run by 181 tests, including 0 in the `bav` library and 53 in the `DifferentialAlgebra` package.

`void bav_term_at_point_int_p (ba0_mpz_t res, struct bav_term *T, struct ba0_point_int_p *point)` [Function]

Assign T evaluated at *point* to *res*. Exception `BA0_ERRALG` is raised if not all variables get evaluated. This function belongs to `bav_term.c`. It is run by 128 tests, including 1 in the `bav` library and 26 in the `DifferentialAlgebra` package.

`void * bav_scanf_term (void *U)` [Function]
The parsing function for terms. It is called by `ba0_scanf/%term`. It may raise exception `BAV_ERRTER`. This function belongs to `bav_term.c`. It is run by 95 tests, including 21 in the `bav` library and 17 in the `DifferentialAlgebra` package.

`void bav_printf_term (void *U)` [Function]
The printing function for terms. It is called by `ba0_printf/%term`. This function belongs to `bav_term.c`. It is run by 328 tests, including 21 in the `bav` library and 59 in the `DifferentialAlgebra` package.

5.16 Term orderings

Three term orderings are implemented: the `lex`, the `grlex` and the `degrevlex` orderings. These terms orderings rely on the current ordering on variables. The default term ordering is the `lex` one. A change of term ordering does not affect the internal representation of terms. It only modifies the behaviour of the term comparison functions. The current term ordering is defined in the variable `bav_global.term_ordering`.

`void bav_set_term_ordering (char *ordering)` [Function]
Set the current term ordering to *ordering* which must be one of the strings `"lex"`, `"grlex"` and `"degrevlex"`. This function belongs to `bav_term_ordering.c`. It is run by 494 tests, including 102 in the `bav` library and 65 in the `DifferentialAlgebra` package.

`enum ba0_compare_code bav_compare_term (struct bav_term *T, struct bav_term *U)` [Function]
Return `ba0_lt` if $T < U$, `ba0_eq` if $T = U$ and `ba0_gt` if $T > U$ w.r.t. the current term ordering. This function belongs to `bav_term_ordering.c`. It is run by 364 tests, including 1 in the `bav` library and 59 in the `DifferentialAlgebra` package.

`enum ba0_compare_code bav_compare_stripped_term (struct bav_term *T, struct bav_term *U, bav_Inumber last_number)` [Function]
Variant of the above function such that only variable whose number (w.r.t. the current ordering on variables) is greater than *last_number* are taken into account. This function belongs to `bav_term_ordering.c`. It is run by 317 tests, including 0 in the `bav` library and 43 in the `DifferentialAlgebra` package.

5.17 Points

`bool bav_is_differentially_ambiguous_point (struct ba0_point *point)` [Function]
Return `true` if one of the variables of *point* is a derivative of some other variable of *point*. This function belongs to `bav_point.c`. It is not run by any test.

`void bav_delete_independent_values_point (struct ba0_point *R, struct ba0_point *P)` [Function]
Remove from P all values with independent `var` field. Result in R. This function belongs to `bav_point.c`. It is not run by any test.

6 The bap library

6.1 Introduction

The bap library mostly aims at implementing basic algorithms for differential polynomials over the GMP big integers (mpz_t, ba0_mpz_t). It provides the basic tools needed for the baz library which focuses on the multivariate gcd and factorization. The bap library provides also variants of these algorithms for polynomials over big rational numbers (mpq_t, ba0_mpq_t), big integer modulo a big integer ba0_mpzm_t and small integers modulo a small number ba0_mint_hp. The bap library relies on the ba0 and bav libraries.

Many identifiers of functions, types, variables ... of the bap library terminate by _COEFF. These are generic functions. At compilation time, the makefile instantiate them replacing the suffix by _mpz (for polynomials with mpz_t coefficients), _mint_hp (for polynomials with modular machine numbers coefficients), _mpzm (for polynomials with big modular numbers coefficients) and _mpq.

6.1.1 Global variables and settings

There are no global variables in the bap library.

`void bap_reset_all_settings (void)` [Function]

Reset all settings variables to their default values. This function must be called outside sequences of calls to the library. This function belongs to bap_common.c. It is run by 65 tests, including 0 in the bap library and 65 in the DifferentialAlgebra package.

6.1.2 Basic types, constants and naming conventions

The name of every exported object defined in the library starts by bap_. Functions starting by bap_init initialize raw data structures (constructors). Functions starting by bap_new dynamically allocate data structures and apply the corresponding bap_init functions on them. Functions starting by bap_reset reset data structures which are assumed to be already initialized. Functions starting by bap_set perform assignments and systematically copy the sources to the destinations. A *sequence of calls* to the bap library starts with a call to bap_restart and finishes by a call to bap_terminate.

`void bap_restart (ba0_int_p` *time_limit*`, ba0_int_p` *memory_limit*`)` [Function]

Call the bav_restart function with the same parameters and defines a few more formats. This function belongs to bap_common.c. It is run by 392 tests, including 33 in the bap library and 65 in the DifferentialAlgebra package.

`void bap_terminate (enum ba0_restart_level` *level*`)` [Function]

Call the bav_terminate function with the same parameter. This function belongs to bap_common.c. It is run by 392 tests, including 33 in the bap library and 65 in the DifferentialAlgebra package.

6.2 Internal representation of polynomials

The data structure essentially is an expanded form. Monomials are sorted by decreasing order w.r.t. the lexicographic ordering induced by the current ordering. The data structure is designed to provide access to coefficients of polynomials with respect to any variable with respect to any ordering without rebuilding the polynomial. Polynomials are built over a simpler data type, called *clot*, which are ordered linear combinations of terms.

6.2.1 Ordered linear combinations of terms

6.2.1.1 Zipped terms

Terms are stored in zipped (compressed) form in polynomials. The type `bap_zipterm` is a redefininition of `ba0_int_p`.

6.2.1.2 Term managers

The data type `bap_zipterm` is actually close to `void *`. With not too much effort, it is possible to change the way terms are managed (a basic term or a pointer towards a hash table of terms). Indeed, such variants of implementations were offered in earlier versions of the library but removed to get a lighter code.

Description of type `bap_termanager`. This data type is designed to store the terms of a polynomial (more precisely, of a clot in compressed form. This type declaration belongs to `bap_termanager.h`.

```
struct bap_termanager
{
// the total rank of all terms of the clot
  struct bav_term total_rank;
  struct _zipping
  {
// number of ba0_int_p allocated to each zipterm
    ba0_int_p alloc;
// number of ba0_int_p used by each zipterm
    ba0_int_p size;
// the number of allocated entries of nbbits and mask
//      is equal to total_rank.alloc
// nbbits[i] = the number of bits needed to store the degree of
//      the ith variable in total_rank in any zipterm
// mask[i]   = the binary mask needed to extract this degree
    unsigned char *nbbits;
    unsigned ba0_int_p *mask;
  } zipping;
};
```

`void bap_init_termanager (struct bap_termanager *`*tgest*`, struct bav_term *`*T*`)` [Function]

Initialize *tgest* with field `total_rank` equal to T. Note that T may be a proper multiple of the actual total rank of the clot. This function belongs to `bap_termanager.c`. It is run by 385 tests, including 32 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_reset_termanager (struct bap_termanager *tgest, struct bav_term *T, bool *reinit)` [Function]

Resets *tgest* with field `total_rank` equal to T. Set *reinit* to `true` if it is necessary to apply `bap_init_zipterm_array_termanager` to all arrays of zipterms formerly handled by this termanager. This function belongs to `bap_termanager.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_init_zipterm_array_termanager (struct bap_termanager *tgest, bap_zipterm *tab, ba0_int_p n)` [Function]

Allocate each entry of the array *tab* of n zipterms so that *tab* can receive the zipterms handled by *tgest*. This function belongs to `bap_termanager.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`bool bap_equal_termanager (struct bap_termanager *A, struct bap_termanager *B)` [Function]

Return `true` if A and B have the same fields `total_rank` so that their zipterms are compatible. This function belongs to `bap_termanager.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_init_set_termanager (struct bap_termanager *dst, struct bap_termanager *src)` [Function]

Assign *src* to *dst*. This function belongs to `bap_termanager.c`. It is run by 265 tests, including 6 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_set_zipterm_zipterm_termanager (struct bap_termanager *dst_tgest, bap_zipterm *dst_zipterm, struct bap_termanager *src_tgest, bap_zipterm src_zipterm)` [Function]

Assign *src_zipterm* (handled by *src_tgest*) to *dst_zipterm* (handled by *dst_tgest*) in the special case where zipterms are compatible. This function belongs to `bap_termanager.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_set_zipterm_term_termanager (struct bap_termanager *tgest, bap_zipterm *zipterm, struct bav_term *T)` [Function]

Compress T and store it in *zipterm*, which is handled by *tgest*. This function belongs to `bap_termanager.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_set_term_zipterm_termanager (struct bap_termanager *tgest, struct bav_term *T, bap_zipterm zipterm)` [Function]

Uncompress *zipterm*, which is handled by *tgest* and store the result in T. This function belongs to `bap_termanager.c`. It is run by 382 tests, including 31 in the `bap` library and 63 in the `DifferentialAlgebra` package.

`bool bap_worth_garbage_zipterm_termanager (struct bap_termanager *tgest)` [Function]

Return `true` if the zipterms handled by *tgest* are allocated areas, which thus need to be processed by the garbage collector. This function belongs to `bap_termanager.c`. It is run by 39 tests, including 7 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_switch_ring_termanager (struct bap_termanager *tgest, struct bav_differential_ring *R)` [Function]

Apply `bav_switch_ring_term` to the `total_rank` field of *tgest*. This function belongs to `bap_termanager.c`. It is run by 38 tests, including 1 in the `bap` library and 36 in the `DifferentialAlgebra` package.

6.2.1.3 Clots

Description of type `bap_table2of_monom_COEFF`. This data type implements a contiguous linear combination of zipterms. This type declaration belongs to `bap_clot_COEFF.h`.

```
struct bap_table2of_monom_COEFF
{
  ba0_int_p alloc;              // number of allocated entries for both arrays
  ba0_int_p size;               // number of used entries for both arrays
  bap_zipterm *zipterm;         // the array of zipterms
  COEFF_t *coeff;               // the array of coefficients
};
```

Description of type `bap_tableof_table2of_monom_COEFF`. This data type is a table of tables. However, if we forget technical details, it is a sequence of pairs of the form (term, coefficient). These pairs are called *monomials* in the documentation. This type declaration belongs to `bap_clot_COEFF.h`.

```
struct bap_tableof_table2of_monom_COEFF
{
  ba0_int_p alloc;              // the number of allocated entries of tab
  ba0_int_p size;               // the number of used entries in tab
  struct bap_table2of_monom_COEFF **tab;
};
```

Description of type `bap_clot_COEFF`. This data type implements a clot i.e. a sequence of monomials sorted in decreasing order with respect to the lexicographic ordering induced by the current ordering. The coefficients of monomials are nonzero. Monomials are sometimes considered as numbered starting from 0 in the documentation. This type declaration belongs to `bap_clot_COEFF.h`.

```
struct bap_clot_COEFF
{
// the total number of allocated entries of (zipterm, coeff)
  ba0_int_p alloc;
// the total number of used entries i.e. of monomials
  ba0_int_p size;
// the termanager which handles zipterms
  struct bap_termanager tgest;
// the table of bap_table2of_monom_COEFF
  struct bap_tableof_table2of_monom_COEFF tab;
// the ordering with respect to which zipterms are ordered
  bav_Iordering ordering;
};
```

`struct bap_clot_COEFF * bap_new_clot_COEFF (struct bav_term *T)` [Function]

Allocate a clot, initialize it and return it. The term T is supposed to be a multiple of all terms present in the clot. This function belongs to `bap_clot_COEFF.c`. It is run by 385 tests, including 32 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`bool bap_is_zero_clot_COEFF (struct bap_clot_COEFF *C)` [Function]

Return `true` if C is zero i.e. if C is empty since monomial coefficients are supposed to be nonzero. This function belongs to `bap_clot_COEFF.c`. It is run by 60 tests, including 0 in the `bap` library and 4 in the `DifferentialAlgebra` package.

`void bap_reverse_clot_COEFF (struct bap_clot_COEFF *C)` [Function]

Revert the sequence of monomials of C. This low-level function is used by the few functions which build clot in the (wrong) ascending order. This function belongs to `bap_clot_COEFF.c`. It is run by 57 tests, including 0 in the `bap` library and 4 in the `DifferentialAlgebra` package.

`void bap_change_ordering_clot_COEFF (struct bap_clot_COEFF *C, bav_Iordering r)` [Function]

Assign r to the field `ordering` of C. This low-level function is used by `bap_physort_polynom_COEFF` only. This function belongs to `bap_clot_COEFF.c`. It is run by 265 tests, including 6 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_sort_clot_COEFF (struct bap_clot_COEFF *clot, ba0_int_p l, ba0_int_p r)` [Function]

Sorts the sequence of monomials of *clot* which have numbers $l \leq n < r$ by decreasing order, according to the lexicographic ordering induced by the ordering. This low-level function is used by `bap_physort_polynom_COEFF`. This function belongs to `bap_clot_COEFF.c`. It is run by 265 tests, including 6 in the `bap` library and 39 in the `DifferentialAlgebra` package.

6.2.1.4 Iterators of monomials over a clot

Description of type `bap_composite_number`. This data structure aims at numbering elements which belong to a sequence of elements implemented by means of tables of tables. Assume an element lies in `T->tab[i]->tab[j]`. Then `primary` is equal to i, `secondary` is equal to j and `combined`, the index of the element in the sequence, is equal to j plus the sum, for $0 \leq k < i$ of `T->tab[k]->size`. This type declaration belongs to `bap_common.h`.

```
struct bap_composite_number
{
  ba0_int_p primary;
  ba0_int_p secondary;
  ba0_int_p combined;
};
```

Description of type `bap_itermon_clot_COEFF`. This data type implements iterators of monomials over a clot. In this context the monomial number `num.combined` is located in `clot->tab[num.primary]->tab[num.secondary]`. An iterator may be set outside a clot. This type declaration belongs to `bap_clot_COEFF.h`.

```
struct bap_itermon_clot_COEFF
```

```
{
  struct bap_clot_COEFF *clot;
  struct bap_composite_number num;
};
```

`void bap_begin_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *iter, struct bap_clot_COEFF *clot)` [Function]

Set *iter* on the first monomial of *clot*. If *clot* is empty, *iter* is set outside the *clot*. This function belongs to `bap_clot_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_end_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *iter, struct bap_clot_COEFF *clot)` [Function]

Set *iter* on the last monomial of *clot*. If *clot* is empty, *iter* is set outside the *clot*. This function belongs to `bap_clot_COEFF.c`. It is run by 57 tests, including 0 in the `bap` library and 4 in the `DifferentialAlgebra` package.

`bool bap_outof_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *iter)` [Function]

Return `true` if *iter* is set outside its clot. This function belongs to `bap_clot_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_next_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *iter)` [Function]

Set *iter* on the next monomial of its clot. Exception `BA0_ERRALG` is raised if *iter* was outside its clot before the call. This function belongs to `bap_clot_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_prev_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *iter)` [Function]

Set *iter* on the previous monomial of its clot. Exception `BA0_ERRALG` is raised if *iter* was outside its clot before the call. This function belongs to `bap_clot_COEFF.c`. It is run by 299 tests, including 13 in the `bap` library and 40 in the `DifferentialAlgebra` package.

`ba0_int_p bap_number_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *iter)` [Function]

Return the number of the current monomial of *iter*. This function belongs to `bap_clot_COEFF.c`. It is run by 382 tests, including 31 in the `bap` library and 63 in the `DifferentialAlgebra` package.

`void bap_term_itermon_clot_COEFF (struct bav_term *T, struct bap_itermon_clot_COEFF *iter)` [Function]

Assign to T the current term of *iter*. Exception `BA0_ERRALG` is raised if *iter* is outside its clot. This function belongs to `bap_clot_COEFF.c`. It is run by 382 tests, including 31 in the `bap` library and 63 in the `DifferentialAlgebra` package.

`COEFF_t * bap_coeff_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *iter)` [Function]

Return the address of the current coefficient of *iter*. Exception `BA0_ERRALG` is raised if *iter* is outside its clot. This function belongs to `bap_clot_COEFF.c`. It is run by 383 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_swap_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *I, struct bap_itermon_clot_COEFF *J)` [Function]

Swap the current monomials of I and J. Exception `BA0_ERRALG` is raised if I or J is outside its clot. This function belongs to `bap_clot_COEFF.c`. It is run by 157 tests, including 5 in the `bap` library and 28 in the `DifferentialAlgebra` package.

`void bap_goto_itermon_clot_COEFF (struct bap_itermon_clot_COEFF *iter, ba0_int_p numero)` [Function]

Set *iter* to the monomial number *numero* of its clot. Exception `BA0_ERRALG` is raised if *numero* is not the number of some monomial of the clot. This function belongs to `bap_clot_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

6.2.1.5 Creators of clots

`void bap_begin_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea, struct bap_clot_COEFF *clot, struct bav_term *T, ba0_int_p table2of_monom_alloc)` [Function]

Set *crea* at the beginning of *clot*. The term manager of *clot* is set to T, which is a multiple of all terms which are going to be stored in *clot*. If some new arrays of type `struct bap_table2of_monom_COEFF *` are required during the creation process, they will have *table2of_monom_alloc* entries. This function belongs to `bap_clot_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_append_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea, struct bap_clot_COEFF *clot, ba0_int_p table2of_monom_alloc)` [Function]

Set *crea* at the end of *clot* so that the newly recorded monomials get appended to *clot*. If some new arrays of type `struct bap_table2of_monom_COEFF *` are required during the creation process, they will have *table2of_monom_alloc* entries. This function belongs to `bap_clot_COEFF.c`. It is run by 4 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_write_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea, struct bav_term *T, COEFF_t c)` [Function]

Write the monomial cT in the clot processed by *crea*. The coefficient c is allowed to be zero (then nothing is done). The creator then moves to the next available monomial entry. This function belongs to `bap_clot_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_write_neg_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea, struct bav_term *T, COEFF_t c)` [Function]

Write the monomial $-cT$ in the clot processed by *crea*. The coefficient c is allowed to be zero (then nothing is done). The creator then moves to the next available monomial entry. This function belongs to `bap_clot_COEFF.c`. It is run by 330 tests, including 18 in the `bap` library and 48 in the `DifferentialAlgebra` package.

`void bap_write_term_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea, struct bav_term *T)` [Function]

Assign T to the current term of the clot processed by *crea*. The coefficient is left unchanged. The creator then moves to the next available monomial entry. This function belongs to `bap_clot_COEFF.c`. It is run by 265 tests, including 6 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_write_all_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea, struct bap_clot_COEFF *clot, ba0_int_p l, ba0_int_p r)` [Function]

Write on the clot processed by *crea* all the monomials of *clot* which have numbers in the range $l \leq n < r$. This function belongs to `bap_clot_COEFF.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_write_neg_all_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea, struct bap_clot_COEFF *clot, ba0_int_p l, ba0_int_p r)` [Function]

Write on the clot processed by *crea* the opposite of all the monomials of *clot* which have numbers in the range $l \leq n < r$ (the signs of the coefficients are changed). This function belongs to `bap_clot_COEFF.c`. It is run by 112 tests, including 13 in the `bap` library and 27 in the `DifferentialAlgebra` package.

`void bap_write_mul_all_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea, struct bap_clot_COEFF *clot, COEFF_t c, ba0_int_p l, ba0_int_p r)` [Function]

Write on the clot processed by *crea* all the monomials of *clot* which have numbers in the range $l \leq n < r$ multiplied by c. If c is zero, nothing is done. This function belongs to `bap_clot_COEFF.c`. It is run by 259 tests, including 16 in the `bap` library and 35 in the `DifferentialAlgebra` package.

`void bap_write_exquo_all_creator_clot_mpz (struct bap_creator_clot_mpz *crea, struct bap_clot_mpz *clot, ba0_mpz_t c, ba0_int_p l, ba0_int_p r)` [Function]

This function belongs to `bap_clot_COEFF.c`. It is not run by any test.

`void bap_close_creator_clot_COEFF (struct bap_creator_clot_COEFF *crea)` [Function]

Achieve the creation process. The resulting clot is not necessarily sorted. This function belongs to `bap_clot_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

6.2.2 Polynomials

In order to provide access of coefficients w.r.t. any variable and any ordering, some subtypes are needed

6.2.2.1 Sequential accesses

Description of type `bap_sequential_access`. This data type permits to define a subsequence of monomials in a clot. This type declaration belongs to `bap_sequential_access.h`.

```
struct bap_sequential_access
{
// the index of the first monomial of the subsequence
  ba0_int_p first;
// the index of the first monomial following the subsequence
  ba0_int_p after;
};
```

6.2.2.2 Indexed accesses

Description of type `bap_indexed_access`. This data type permits to define a sequence of non consecutive monomials in a clot or in a polynomial, in any order. It is a table of tables of monomial numbers. From a logical point of view, it is a sequence of monomial numbers. This type declaration belongs to `bap_indexed_access.h`.

```
struct bap_indexed_access
{
// the total number of entries allocated to tab
  ba0_int_p alloc;
// the total number of entries used in tab
  ba0_int_p size;
  struct ba0_tableof_tableof_int_p tab;
};
```

`void bap_init_indexed_access (struct bap_indexed_access *ind)` [Function]

Initialize *ind*. This function belongs to `bap_indexed_access.c`. It is run by 385 tests, including 32 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_realloc_indexed_access (struct bap_indexed_access *ind, ba0_int_p nbmon)` [Function]

Reallocate the field `tab` if *ind* so that it can receive *nbmon* entries. This function belongs to `bap_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`void bap_reverse_indexed_access (struct bap_indexed_access *ind)` [Function]

Revert the sequence of indices present in *ind*. This function belongs to `bap_indexed_access.c`. It is not run by any test.

6.2.2.3 Iterators for indexed accesses

`void bap_begin_iterator_indexed_access (struct bap_iterator_indexed_access *iter, struct bap_indexed_access *ind)` [Function]

Set *iter* on the first entry of *ind*. This function belongs to `bap_iterator_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`void bap_end_iterator_indexed_access (struct bap_iterator_indexed_access *iter, struct bap_indexed_access *ind)` [Function]

Set *iter* on the last entry of *ind*. This function belongs to `bap_iterator_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`bool bap_outof_iterator_indexed_access (struct bap_iterator_indexed_access *iter)` [Function]

Return `true` if *iter* is outside its `ind` field. This function belongs to `bap_iterator_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`void bap_next_iterator_indexed_access (struct bap_iterator_indexed_access *iter)` [Function]

Move *iter* to the next entry. Exception `BA0_ERRALG` is raised if *iter* is outside its `ind` field before the call. This function belongs to `bap_iterator_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`void bap_prev_iterator_indexed_access (struct bap_iterator_indexed_access *iter)` [Function]

Move *iter* to the previous entry. Exception `BA0_ERRALG` is raised if *iter* is outside its `ind` field before the call. This function belongs to `bap_iterator_indexed_access.c`. It is run by 236 tests, including 5 in the `bap` library and 31 in the `DifferentialAlgebra` package.

`void bap_goto_iterator_indexed_access (struct bap_iterator_indexed_access *iter, ba0_int_p n)` [Function]

Move *iter* to the *n*th entry. Exception `BA0_ERRALG` is raised if *n* is not a valid entry number. This function belongs to `bap_iterator_indexed_access.c`. It is run by 253 tests, including 10 in the `bap` library and 40 in the `DifferentialAlgebra` package.

`ba0_int_p bap_index_iterator_indexed_access (struct bap_iterator_indexed_access *iter)` [Function]

Return the current index. Exception `BA0_ERRALG` is raised if *iter* is outside its `ind` field before the call. This function belongs to `bap_iterator_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`ba0_int_p bap_read_iterator_indexed_access (struct bap_iterator_indexed_access *iter)` [Function]

Return the current index then move *iter* to the next entry. This function belongs to `bap_iterator_indexed_access.c`. It is run by 241 tests, including 5 in the `bap` library and 31 in the `DifferentialAlgebra` package.

`void bap_swapindex_iterator_indexed_access (struct bap_iterator_indexed_access *I, struct bap_iterator_indexed_access *J)` [Function]

Swap the current indices of I and J. Exception `BA0_ERRALG` is raised if any of the iterators is outside its `ind` field. This function belongs to `bap_iterator_indexed_access.c`. It is run by 200 tests, including 9 in the `bap` library and 38 in the `DifferentialAlgebra` package.

`void bap_set_iterator_indexed_access (struct bap_iterator_indexed_access *I, struct bap_iterator_indexed_access *J)` [Function]

Assign J to I. This function belongs to `bap_iterator_indexed_access.c`. It is run by 276 tests, including 12 in the `bap` library and 40 in the `DifferentialAlgebra` package.

6.2.2.4 Creators of indexed accesses

Description of type `bap_creator_indexed_access`. This data structure permits to rewrite the content of a `bap_indexed_access` structure, provided that the already allocated tables are large enough. This type declaration belongs to `bap_iterator_indexed_access.h`.

```
struct bap_creator_indexed_access
{
  struct bap_indexed_access *ind;       // the structure being rewritten
  struct bap_composite_number num;      // the current index
};
```

`void bap_begin_creator_indexed_access (struct bap_creator_indexed_access *crea, struct bap_indexed_access *ind)` [Function]

Set *crea* at the beginning of *ind*. This function belongs to `bap_iterator_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`void bap_write_creator_indexed_access (struct bap_creator_indexed_access *crea, ba0_int_p n)` [Function]

Assign n to the current index of *crea*. Move *crea* to the next available entry of its `ind` field. This function belongs to `bap_iterator_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`void bap_close_creator_indexed_access (struct bap_creator_indexed_access *crea)` [Function]

Complete the creation process. This function belongs to `bap_iterator_indexed_access.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

6.2.2.5 Term strippers

Description of type `bap_termstripper`. This data type is involved in the process of taking coefficients of polynomials with respect to any variable.

It aims at stripping terms i.e. at cutting, in terms, all ranks which depend on variables strictly greater than a given variable. The ordering with respect to which variables are compared is often not the ordering of the clot. A stripping operation is thus described by a pair (`variable`, `ordering`).

The variable is allowed to be `BAV_NOT_A_VARIABLE` (i.e. 0). In that case, all variables are cut. It is allowed to be -1. In that case, no variables are cut.

It happens that a polynomial is obtained by performing a sequence of stripping operations. This sequence, which is necessary to recover the right stripped terms, is recorded. This type declaration belongs to `bap_termstripper.h`.

```
struct bap_termstripper
{
// the number of stripping operations recorded in tab
  ba0_int_p size;
  struct _tab
  {
    bav_Iordering ordering;
    struct bav_variable *varmax;
  } tab[BAP_TERMSTRIPPER_SIZE];
};
```

`void bap_init_set_termstripper (struct bap_termstripper *s, struct bav_variable *v, bav_Iordering r)` [Function]

Initialize s with v and r. This function belongs to `bap_termstripper.c`. It is run by 385 tests, including 32 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_set_termstripper (struct bap_termstripper *dst, struct bap_termstripper *src)` [Function]

Assign *src* to *dst*. This function belongs to `bap_termstripper.c`. It is run by 325 tests, including 16 in the `bap` library and 44 in the `DifferentialAlgebra` package.

`void bap_change_ordering_termstripper (struct bap_termstripper *s, bav_Iordering r)` [Function]

Assign r to the ordering of the last recorded stripping operation. Assign v to the variable of the last recorded stripping operation. This function belongs to `bap_termstripper.c`. It is run by 265 tests, including 6 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_change_variable_termstripper (struct bap_termstripper *s, struct bav_variable *v)` [Function]

This function belongs to `bap_termstripper.c`. It is run by 314 tests, including 11 in the `bap` library and 43 in the `DifferentialAlgebra` package.

`void bap_append_termstripper (struct bap_termstripper *s, struct bav_variable *v, bav_Iordering r)` [Function]

Record the new stripping operation defined by r and v. This function belongs to `bap_termstripper.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`void bap_strip_term_termstripper (struct bav_term *T, bav_Iordering r, struct bap_termstripper *s)` [Function]

Apply the stripping operations defined by s to the term T. The result is in T and is sorted according to the ordering r. This function belongs to `bap_termstripper.c`. It is run by 382 tests, including 31 in the `bap` library and 63 in the `DifferentialAlgebra` package.

`bool bap_identity_termstripper (struct bap_termstripper *s, bav_Iordering r)` [Function]

Return `true` if the stripping operations defined by s are the identity. This function belongs to `bap_termstripper.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_switch_ring_termstripper (struct bap_termstripper *s, struct bav_differential_ring *R)` [Function]

Apply `bav_switch_ring_variable` to all variables in s. This function belongs to `bap_termstripper.c`. It is run by 38 tests, including 1 in the `bap` library and 36 in the `DifferentialAlgebra` package.

6.2.2.6 Polynomials

Description of type `bap_polynom_COEFF`. This data type implements polynomials. Note that the `total_rank` field contains the exact total rank of the polynomial while the `total_rank` field of the term manager of its clot only contains a multiple of this total rank. This type declaration belongs to `bap_polynom_COEFF.h`.

```
struct bap_polynom_COEFF
{
  struct bap_clot_COEFF *clot;  // the underlying clot
  struct bav_term total_rank;   // the total rank of the polynomial
  bool readonly;                // true if modifications are forbidden
  enum bap_typeof_monom_access access;
  struct bap_sequential_access seq;     // in case of a sequential access
  struct bap_indexed_access ind;        // in case of an indexed access
  struct bap_termstripper tstrip;       // for stripping terms
};
```

6.3 Primitives for polynomials

6.3.1 Miscellaneous low level primitives

`void bap_set_total_rank_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

This low-level function computes and sets the `total_rank` field of A. This function belongs to `bap_polynom_COEFF.c`. It is run by 377 tests, including 30 in the `bap` library and 61 in the `DifferentialAlgebra` package.

`void bap_reverse_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

This low-level function reverts the order of the monomials in the clot of A. Exception `BA0_ERRALG` is raised if A has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 57 tests, including 0 in the `bap` library and 4 in the `DifferentialAlgebra` package.

`void bap_physort_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

This low-level function sorts the polynomial A with respect to the current ordering. The polynomial A is supposed to have sequential access. This function belongs to `bap_polynom_COEFF.c`. It is run by 265 tests, including 6 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bap_sizeof_polynom_COEFF (struct bap_polynom_COEFF *F, enum ba0_garbage_code code)` [Function]

Return the number of bytes used to store F. If *code* is `ba0_embedded` then F is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `bap_polynom_COEFF.c`. It is run by 38 tests, including 1 in the `bap` library and 36 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bap_sizeof_tableof_polynom_COEFF (struct bap_tableof_polynom_COEFF *T, enum ba0_garbage_code code)` [Function]

Return the number of bytes used to store T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`void bap_switch_ring_polynom_COEFF (struct bap_polynom_COEFF *A, struct bav_differential_ring *R)` [Function]

Apply `bav_switch_ring_variable` over all the variables occurring in A. The polynomial A is modified. This low level function should be used in conjunction with `bav_set_differential_ring`: if R is a ring obtained by application of `bav_set_differential_ring` to the ring A refers to, then this function transforms A as a polynomial of R. This function belongs to `bap_polynom_COEFF.c`. It is run by 38 tests, including 1 in the `bap` library and 36 in the `DifferentialAlgebra` package.

`bool bap_are_disjoint_polynom_COEFF (struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)` [Function]

Return `true` if A and B are disjoint, in the sense that any modification on one of these polynomials has no side effect on the other one. This function belongs to `bap_polynom_COEFF.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_sort_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A)` [Function]

Assign to R the polynomial A, sorted with respect to the current ordering. The resulting polynomial R is readonly and has indexed access. The polynomial A is left

unchanged. This function belongs to `bap_polynom_COEFF.c`. It is run by 280 tests, including 12 in the `bap` library and 41 in the `DifferentialAlgebra` package.

6.3.2 Initialization and assignment primitives

`void bap_init_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]
Initialize A to 0. This function belongs to `bap_polynom_COEFF.c`. It is run by 384 tests, including 32 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_init_readonly_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]
Initialize A to 0 as a readonly polynomial. This function belongs to `bap_polynom_COEFF.c`. It is run by 349 tests, including 14 in the `bap` library and 54 in the `DifferentialAlgebra` package.

`void bap_init_polynom_one_COEFF (struct bap_polynom_COEFF *A)` [Function]
Initialize A to 1. This function belongs to `bap_polynom_COEFF.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_init_polynom_crk_COEFF (struct bap_polynom_COEFF *A, COEFF_t c, struct bav_rank *rg)` [Function]
Initialize A to $c\,rg$. The coefficient c is allowed to be 0. The rank rg is allowed to be the rank of zero. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`struct bap_polynom_COEFF * bap_new_polynom_COEFF (void)` [Function]
Allocate a new polynomial, initialize it and return it. This function belongs to `bap_polynom_COEFF.c`. It is run by 379 tests, including 30 in the `bap` library and 62 in the `DifferentialAlgebra` package.

`struct bap_polynom_COEFF * bap_new_readonly_polynom_COEFF (void)` [Function]
Allocate a new polynomial, initialize it as a readonly polynomial and return it. This function belongs to `bap_polynom_COEFF.c`. It is run by 269 tests, including 5 in the `bap` library and 38 in the `DifferentialAlgebra` package.

`struct bap_polynom_COEFF * bap_new_polynom_crk_COEFF (COEFF_t c, struct bav_rank *rg)` [Function]
Allocate a new polynomial, initialize it to $c\,rg$ and return it. The coefficient c is allowed to be 0. The rank rg is allowed to be the rank of zero. This function belongs to `bap_polynom_COEFF.c`. It is run by 5 tests, including 1 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`struct bap_polynom_COEFF * bap_new_polynom_one_COEFF (void)` [Function]
Allocate a new polynomial, initialize it to 1 and return it. This function belongs to `bap_polynom_COEFF.c`. It is run by 5 tests, including 1 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_set_polynom_zero_COEFF (struct bap_polynom_COEFF *A)` [Function]

Assign 0 to A. The polynomial A is allowed to be readonly. This function belongs to `bap_polynom_COEFF.c`. It is run by 348 tests, including 26 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_set_polynom_one_COEFF (struct bap_polynom_COEFF *A)` [Function]

Assign 1 to A. Exception `BAO_ERRALG` is raised if A has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 318 tests, including 4 in the `bap` library and 62 in the `DifferentialAlgebra` package.

`void bap_set_polynom_crk_COEFF (struct bap_polynom_COEFF *A, COEFF_t c, struct bav_rank *rg)` [Function]

Assign $c\,rg$ to A. The coefficient c is allowed to be 0. The rank rg is allowed to be the rank of zero. Note: the clot of A is modified even if the the result is zero. Exception `BAO_ERRALG` is raised if A has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 365 tests, including 27 in the `bap` library and 55 in the `DifferentialAlgebra` package.

`void bap_set_polynom_variable_COEFF (struct bap_polynom_COEFF *A, struct bav_variable *v, bav_Idegree d)` [Function]

Assign v^d to A. Exception `BAO_ERRALG` is raised if A has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 380 tests, including 28 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_set_polynom_term_COEFF (struct bap_polynom_COEFF *A, struct bav_term *T)` [Function]

Assign T to A. Exception `BAO_ERRALG` is raised if A has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 1 tests, including 0 in the `bap` library and 1 in the `DifferentialAlgebra` package.

`void bap_set_polynom_monom_COEFF (struct bap_polynom_COEFF *A, COEFF_t c, struct bav_term *T)` [Function]

Assign $c\,T$ to A. The coefficient c is allowed to be 0. Exception `BAO_ERRALG` is raised if A has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 93 tests, including 0 in the `bap` library and 9 in the `DifferentialAlgebra` package.

`void bap_set_readonly_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A)` [Function]

Assign A to R. The polynomial R is readonly. The two polynomials are not disjoint. This function belongs to `bap_polynom_COEFF.c`. It is run by 214 tests, including 3 in the `bap` library and 38 in the `DifferentialAlgebra` package.

`void bap_set_polynom_COEFF (struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)` [Function]

Assign B to A. The two polynomials are supposed to be disjoint. Exception `BAO_ERRALG` is raised if A has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_set_tableof_polynom_COEFF (struct bap_tableof_polynom_COEFF *dst, struct bap_tableof_polynom_COEFF *src)` [Function]

Assign *src* to *dst*. Entries of *src* are allowed to be `BAP_NOT_A_POLYNOM_COEFF`. This function belongs to `bap_polynom_COEFF.c`. It is run by 38 tests, including 0 in the `bap` library and 8 in the `DifferentialAlgebra` package.

`void bap_set_tableof_tableof_polynom_COEFF (struct bap_tableof_tableof_polynom_COEFF *dst, struct bap_tableof_tableof_polynom_COEFF *src)` [Function]

Assign *src* to *dst*. All tables are allowed to involve zero entries. This function belongs to `bap_polynom_COEFF.c`. It is run by 38 tests, including 0 in the `bap` library and 8 in the `DifferentialAlgebra` package.

`void bap_set_tableof_tableof_tableof_polynom_COEFF (struct bap_tableof_tableof_tableof_polynom_COEFF *dst, struct bap_tableof_tableof_tableof_polynom_COEFF *src)` [Function]

Assign *src* to *dst*. All tables are allowed to involve zero entries. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

6.3.3 Predicates

`bool bap_is_zero_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return `true` if *A* is equal to 0. This function belongs to `bap_polynom_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`bool bap_is_one_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return `true` if *A* is equal to 1. This function belongs to `bap_polynom_COEFF.c`. It is run by 274 tests, including 0 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`bool bap_is_numeric_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return `true` if *A* does not depend on any variable. This function belongs to `bap_polynom_COEFF.c`. It is run by 381 tests, including 30 in the `bap` library and 63 in the `DifferentialAlgebra` package.

`bool bap_is_univariate_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return `true` if *A* depends on a single variable. This function belongs to `bap_polynom_COEFF.c`. It is run by 210 tests, including 0 in the `bap` library and 33 in the `DifferentialAlgebra` package.

`bool bap_is_variable_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return `true` if *A* is equal to a variable. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`void bap_numeric_initial_one_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A)` [Function]

This function belongs to `bap_invert_COEFF.c`. It is run by 136 tests, including 0 in the `bap` library and 23 in the `DifferentialAlgebra` package.

`bool bap_is_solved_numeric_initial_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return `true` if A has leading degree 1 and a numeric initial. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`bool bap_is_solved_constant_initial_polynom_COEFF (struct bap_polynom_COEFF *A, struct bav_symbol *x)` [Function]

Return `true` if A has leading degree 1 and an initial constant with respect to x. This function belongs to `bap_polynom_COEFF.c`. It is run by 26 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`bool bap_is_derivative_minus_independent_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return `true` if A has the form $v - B$ where v is a derivative and B is a polynomial which does not depend on any derivative. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`bool bap_is_rank_minus_monom_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

This function belongs to `bap_polynom_COEFF.c`. It is run by 152 tests, including 0 in the `bap` library and 27 in the `DifferentialAlgebra` package.

`bool bap_depend_polynom_COEFF (struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Return `true` if A depends on v. This function belongs to `bap_polynom_COEFF.c`. It is run by 252 tests, including 3 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`bool bap_depend_only_polynom_COEFF (struct bap_polynom_COEFF *A, struct bav_tableof_variable *T)` [Function]

Return `true` if A depends only on the variables present in T. The table T is supposed to be sorted in ascending order. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

6.3.4 Comparing and sorting

`enum ba0_compare_code bap_compare_polynom_COEFF (struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)` [Function]

Enumerate all the monomials of A and B and looks for the first terms which differ. Return `ba0_lt` if the term of A is strictly less than that of B, `ba0_gt` if the term of A is strictly greater than that of B, `ba0_eq` if A and B are equal, and and `ba0_equiv` A and B have the same terms but not the same coefficients. This function belongs to `bap_polynom_COEFF.c`. It is run by 131 tests, including 0 in the `bap` library and 26 in the `DifferentialAlgebra` package.

`void bap_sort_tableof_polynom_COEFF (struct bap_tableof_polynom_COEFF *T, enum ba0_sort_mode mode)` [Function]

Sort T in ascending or descending order depending on *mode*. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`bool bap_equal_polynom_COEFF (struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)` [Function]

Return `true` if A and B are equal. This function belongs to `bap_polynom_COEFF.c`. It is run by 331 tests, including 8 in the `bap` library and 57 in the `DifferentialAlgebra` package.

`bool bap_gt_rank_polynom_COEFF (void *P, void *Q)` [Function]

Return `true` if P has strictly greater rank than B. Both parameters are polynomials. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`bool bap_lt_rank_polynom_COEFF (void *P, void *Q)` [Function]

Return `true` if P has strictly lower rank than B. Both parameters are polynomials. This function belongs to `bap_polynom_COEFF.c`. It is run by 249 tests, including 0 in the `bap` library and 38 in the `DifferentialAlgebra` package.

`bool bap_equal_rank_polynom_COEFF (struct bap_polynom_COEFF *P, struct bap_polynom_COEFF *Q)` [Function]

Return `true` if P and Q have the same rank. This function belongs to `bap_polynom_COEFF.c`. It is run by 50 tests, including 0 in the `bap` library and 16 in the `DifferentialAlgebra` package.

`int bap_compare_rank_polynom_COEFF (const void *PP, const void *QQ)` [Function]

This function belongs to `bap_polynom_COEFF.c`. It is run by 120 tests, including 0 in the `bap` library and 12 in the `DifferentialAlgebra` package.

6.3.5 Degree, rank, initial, separant

`ba0_int_p bap_nbmon_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return the number of monomials present in A. This function belongs to `bap_polynom_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`struct bav_variable * bap_leader_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return the leading variable of A. Exception `BAP_ERRCST` is raised if A does not depend on any variable. This function belongs to `bap_polynom_COEFF.c`. It is run by 307 tests, including 6 in the `bap` library and 48 in the `DifferentialAlgebra` package.

`bav_Idegree bap_degree_polynom_COEFF (struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Return the degree of A with respect to v. Return -1 if A is 0. This function belongs to `bap_polynom_COEFF.c`. It is run by 241 tests, including 5 in the `bap` library and 35 in the `DifferentialAlgebra` package.

`struct bav_rank bap_rank_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return the leading rank of A. This function belongs to `bap_polynom_COEFF.c`. It is run by 284 tests, including 1 in the `bap` library and 43 in the `DifferentialAlgebra` package.

`void bap_minimal_total_rank_polynom_COEFF (struct bav_term *T, struct bap_polynom_COEFF *A)` [Function]

Assign to T the gcd of all the terms occurring in A. Exception `BAP_ERRNUL` is raised if A is 0. This function belongs to `bap_polynom_COEFF.c`. It is run by 331 tests, including 0 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`bav_Idegree bap_leading_degree_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return the degree of A with respect to its leading variable. Exception `BAP_ERRCST` is raised if A does not depend on any variable. This function belongs to `bap_polynom_COEFF.c`. It is run by 267 tests, including 2 in the `bap` library and 38 in the `DifferentialAlgebra` package.

`bav_Idegree bap_total_degree_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return the total degree of A. Return -1 if A is 0. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`bav_Iorder bap_total_order_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return the sum of the orders of all the derivatives A depends on. This function belongs to `bap_polynom_COEFF.c`. It is run by 1 tests, including 0 in the `bap` library and 1 in the `DifferentialAlgebra` package.

`void bap_leading_term_polynom_COEFF (struct bav_term *T, struct bap_polynom_COEFF *A)` [Function]

Assign to T the leading term of A. Exception `BAP_ERRNUL` is raised if A is 0. This function belongs to `bap_polynom_COEFF.c`. It is run by 145 tests, including 0 in the `bap` library and 25 in the `DifferentialAlgebra` package.

`COEFF_t * bap_numeric_initial_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return the address of the coefficient of the leading term of A. Exception `BAP_ERRNUL` is raised if A is 0. This function belongs to `bap_polynom_COEFF.c`. It is run by 374 tests, including 27 in the `bap` library and 62 in the `DifferentialAlgebra` package.

`void bap_initial_and_reductum_polynom_COEFF (struct bap_polynom_COEFF *initial, struct bap_polynom_COEFF *reductum, struct bap_polynom_COEFF *A)` [Function]

Assign to *initial* and *reductum* the initial and the reductum of A, in readonly mode. Both parameters are allowed to be the zero pointers. Exception `BAP_ERRCST` is raised if A does not depend on any variable. This function belongs to `bap_polynom_COEFF.c`. It is run by 269 tests, including 6 in the `bap` library and 40 in the `DifferentialAlgebra` package.

`void bap_initial_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A)` [Function]

Assign to *initial* the initial of A, in readonly mode. Exception `BAP_ERRCST` is raised if A does not depend on any variable. This function belongs to `bap_polynom_COEFF.c`. It is run by 232 tests, including 1 in the `bap` library and 35 in the `DifferentialAlgebra` package.

`void bap_reductum_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A)` [Function]

Assign to *reductum* the reductum of A, in readonly mode. Exception `BAP_ERRCST` is raised if A does not depend on any variable. This function belongs to `bap_polynom_COEFF.c`. It is run by 150 tests, including 2 in the `bap` library and 31 in the `DifferentialAlgebra` package.

`void bap_coeff_polynom_COEFF (struct bap_polynom_COEFF *C, struct bap_polynom_COEFF *A, struct bav_variable *v, bav_Idegree d)` [Function]

Assign to C the coefficient of $v\hat{}d$ in A (the leading variable of A is taken if v is `BAV_NOT_A_VARIABLE`). The variable v is not required to be greater than or equal to the leading variable of A. The resulting polynomial is not readonly. It is disjoint from A. Exception `BAO_ERRALG` is raised if it has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 38 tests, including 0 in the `bap` library and 9 in the `DifferentialAlgebra` package.

`void bap_initial_and_reductum2_polynom_COEFF (struct bap_polynom_COEFF *initial, struct bap_polynom_COEFF *reductum, struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Variant of `bap_initial_and_reductum_polynom_COEFF` where the variable v with respect to which the initial and the reductum are defined is specified. This variable is supposed to be greater than or equal to the leader of A. The polynomial A is allowed to be numeric. The resulting polynomials are readonly. This function belongs to `bap_polynom_COEFF.c`. It is run by 221 tests, including 3 in the `bap` library and 35 in the `DifferentialAlgebra` package.

`void bap_initial2_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Variant of `bap_initial_polynom_COEFF`. See `bap_initial_and_reductum2_polynom_COEFF`. This function belongs to `bap_polynom_COEFF.c`. It is run by 107 tests, including 3 in the `bap` library and 12 in the `DifferentialAlgebra` package.

`void bap_reductum2_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Variant of `bap_reductum_polynom_COEFF`. See `bap_initial_and_reductum2_polynom_COEFF`. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`void bap_coeff2_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_variable *v, bav_Idegree d)` [Function]

Assign to R the coefficient of v^d in A. The polynomial R is readonly. The variable v is supposed too be greater than or equal to the leader of A. This function belongs to `bap_polynom_COEFF.c`. It is run by 7 tests, including 2 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`void bap_lcoeff_and_reductum_polynom_COEFF (struct bap_polynom_COEFF *lcoeff, struct bap_polynom_COEFF *reductum, struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Assign to *lcoeff* and *reductum* the leading coefficient and the reductum of A with respect to v (or with respect to the leading variable of A if v is `BAV_NOT_A_VARIABLE`. The variable v is not required to be greater than or equal to the leading variable of A. The resulting polynomials are not readonly. They are disjoint from A. Exception `BAO_ERRALG` is raised if *lcoeff* or *reductum* has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 103 tests, including 2 in the `bap` library and 21 in the `DifferentialAlgebra` package.

`void bap_lcoeff_polynom_COEFF (struct bap_polynom_COEFF *lcoeff, struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Variant of `bap_lcoeff_and_reductum_polynom_COEFF` for the leading coefficient only. This function belongs to `bap_polynom_COEFF.c`. It is run by 101 tests, including 2 in the `bap` library and 19 in the `DifferentialAlgebra` package.

`void bap_coeff_and_reductum_polynom_COEFF (struct bap_polynom_COEFF *C, struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_variable *v, bav_Idegree d)` [Function]

Assign to C the coefficient of $v\hat{}d$ in A (the leading variable of A is taken if v is `BAV_NOT_A_VARIABLE`). Assign to R the polynomial $A - C\,v^d$. The variable v is not required to be greater than or equal to the leading variable of A. The resulting polynomials are not readonly. They are disjoint from A. Exception `BAO_ERRALG` is raised one of them has readonly mode. This function belongs to `bap_polynom_COEFF.c`. It is run by 1 tests, including 1 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_replace_initial_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *C)` [Function]

Assign to R the polynomial obtained by replacing the initial of A by C. Exception `BAP_ERRCST` is raised if A is numeric. xception `BAO_ERRALG` is raised if R is readonly. This function belongs to `bap_polynom_COEFF.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_replace_initial2_polynom_mpz (struct bap_polynom_mpz *R, struct bap_product_mpz *U, struct bap_product_mpz *G, struct bap_polynom_mpz *A)` [Function]

Assign to R the polynomial obtained by replacing the initial of $U\,A$ by G. This function is used for normalizing regular chain polynomials. In this context, U is

the algebraic inverse of the initial of A. Exception `BAP_ERRCST` is raised if A is numeric. Exception `BAO_ERRALG` is raised if R is readonly. This function belongs to `bap_polyspec_mpz.c`. It is run by 25 tests, including 0 in the `bap` library and 13 in the `DifferentialAlgebra` package.

`void bap_separant_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A)` [Function]

Assign to R the separant of A. Exception `BAP_ERRCST` is raised if A is numeric. Exception `BAO_ERRALG` is raised if R is readonly. This function belongs to `bap_polynom_COEFF.c`. It is run by 239 tests, including 2 in the `bap` library and 36 in the `DifferentialAlgebra` package.

`void bap_separant2_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Assign to R the separant of A with respect to v. Exception `BAP_ERRCST` is raised if A is numeric. xception `BAO_ERRALG` is raised if R is readonly. This function belongs to `bap_polynom_COEFF.c`. It is run by 215 tests, including 0 in the `bap` library and 35 in the `DifferentialAlgebra` package.

`void bap_separant_and_sepuctum_polynom_mpz (struct bap_polynom_mpz *S, struct bap_polynom_mpz *R, struct bap_polynom_mpz *A)` [Function]

Assign to S the separant of A. Assign to R the polynomial $d\,A - v\,S$ where v^d denotes the rank of A. The argument S is allowed to be zero. Exception `BAP_ERRCST` is raised if A is numeric. Exception `BAO_ERRALG` is raised if S or R is readonly. This function belongs to `bap_polyspec_mpz.c`. It is run by 135 tests, including 1 in the `bap` library and 30 in the `DifferentialAlgebra` package.

`void bap_sepuctum_polynom_mpz (struct bap_polynom_mpz *R, struct bap_polynom_mpz *A)` [Function]

Assign to R the polynomial $d\,A - v\,S$ where v^d and S denote the rank and the separant of A. Exception `BAP_ERRCST` is raised if A is numeric. Exception `BAO_ERRALG` is raised if R is readonly. This function belongs to `bap_polyspec_mpz.c`. It is run by 1 tests, including 1 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_involved_derivations_polynom_COEFF (struct bav_tableof_variable *T, struct bap_polynom_COEFF *P)` [Function]

Append to T the derivations involved in the derivatives P depends on. The independent variables occurring in P are not taken into account. This function belongs to `bap_diff_polynom_COEFF.c`. It is run by 139 tests, including 0 in the `bap` library and 30 in the `DifferentialAlgebra` package.

`void bap_involved_parameters_polynom_COEFF (struct bav_dictionary_symbol *dict, struct bav_tableof_symbol *T, struct bap_polynom_COEFF *P)` [Function]

Append to T the parameters occurring in P. Every parameter already present in T is supposed to be registered in the dictionary *dict*. This function belongs to `bap_diff_polynom_COEFF.c`. It is run by 139 tests, including 0 in the `bap` library and 30 in the `DifferentialAlgebra` package.

`void bap_split_polynom_COEFF (struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B, struct bap_polynom_COEFF *R, struct bav_term *T)` [Function]

Split R into two parts. Assign to A, as a readonly polynomial, the sum of all the monomials which are greater than or equal to T. Assign to B the sum of the remaining monomials. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 7 tests, including 1 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`void bap_mark_indets_polynom_COEFF (struct bav_dictionary_variable *dict, struct bav_tableof_variable *vars, struct bap_polynom_COEFF *A)` [Function]

Append to *vars* the variables occurring in A which are not already present in *vars*. Every element of *vars* is supposed to be registered in the dictionary *dict*. This function belongs to `bap_polynom_COEFF.c`. It is run by 289 tests, including 3 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_mark_indets_tableof_polynom_COEFF (struct bav_dictionary_variable *dict, struct bav_tableof_variable *vars, struct bap_tableof_polynom_COEFF *T)` [Function]

Append to *vars* the variables occurring in T which are not already present in *vars*. Every element of *vars* is supposed to be registered in the dictionary *dict*. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

`void bap_mark_indets_listof_polynom_COEFF (struct bav_dictionary_variable *dict, struct bav_tableof_variable *vars, struct bap_listof_polynom_COEFF *L)` [Function]

Append to *vars* the variables occurring in L which are not already present in *vars*. Every element of *vars* is supposed to be registered in the dictionary *dict*. This function belongs to `bap_polynom_COEFF.c`. It is not run by any test.

6.3.5.1 Parsing and printing polynomials

Basic formats for polynomials are `Az` (for `ba0_mpz_t` coefficients), `Aq` (for `ba0_mpq_t` coefficients), `Azm` (for modular `ba0_mpz_t` coefficients), `Aim` (for modular `ba0_mint_hp` coefficients).

`void * bap_scanf_polynom_COEFF (void *R)` [Function]

General parser for polynomials. It is called by `ba0_scanf/%A_`. The parsed polynomial is returned and assigned to R if R is not the zero address. Zero derivatives of parameters are not simplified to zero. Exception `BA0_ERRALG` is raised if R is readonly. This function belongs to `bap_parse_polynom_COEFF.c`. It is run by 279 tests, including 30 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void * bap_scanf_simplify_polynom_COEFF (void *A)` [Function]

Another general parser for polynomials. It is called by `ba0_scanf/%simplify_A_`. The parsed polynomial is returned and assigned to R if R is not the zero address. Zero derivatives of parameters are simplified to zero. The fact that derivations commute together is taken into account. Exception `BA0_ERRALG` is raised if R is readonly. This function belongs to `bap_parse_polynom_COEFF.c`. It is run by 33 tests, including 0 in the `bap` library and 33 in the `DifferentialAlgebra` package.

`void * bap_scanf_expanded_polynom_COEFF (void *R)` [Function]
General parser for polynomials in expanded form. It is called by `ba0_scanf/%expanded_A_`. Zero derivatives of parameters are not simplified to zero. The parsed polynomial is returned and assigned to R if R is not the zero address. Exception `BA0_ERRALG` is raised if R is readonly. This function belongs to `bap_parse_polynom_COEFF.c`. It is run by 48 tests, including 2 in the `bap` library and 1 in the `DifferentialAlgebra` package.

`void * bap_scanf_simplify_expanded_polynom_COEFF (void *A)` [Function]
Another general parser for polynomials in expanded form. It is called by `ba0_scanf/%simplify_expanded_A_`. This one simplifies the zero derivatives of parameters occurring in the parsed polynomial and takes into account the fact that derivations commute. The parsed polynomial is returned and assigned to R if R is not the zero address. Exception `BA0_ERRALG` is raised if R is readonly. This function belongs to `bap_parse_polynom_COEFF.c`. It is run by 1 tests, including 1 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void * bap_scanf_polynom_eqn_COEFF (void *R)` [Function]
A polynomial parsing function which parses either a mere polynomial or an equation of the form $p = q$ where p and q are two polynomials. In this case, the parsed polynomial is the difference $p-q$. This function relies on `bap_scanf_polynom_COEFF`. This function belongs to `bap_parse_polynom_COEFF.c`. It is not run by any test.

`void bap_printf_polynom_COEFF (void *R)` [Function]
General printing function for polynomials. It is called by `ba0_printf/%A_`. This function belongs to `bap_parse_polynom_COEFF.c`. It is run by 317 tests, including 27 in the `bap` library and 57 in the `DifferentialAlgebra` package.

6.3.5.2 Addition of polynomials

`void bap_add_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)` [Function]
Assign $A + B$ to R. This function belongs to `bap_add_polynom_COEFF.c`. It is run by 360 tests, including 24 in the `bap` library and 57 in the `DifferentialAlgebra` package.

`void bap_add_polynom_numeric_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, COEFF_t c)` [Function]
Assign $A + c$ to R. This function belongs to `bap_add_polynom_COEFF.c`. It is run by 4 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_sub_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)` [Function]
Assign $A - B$ to R. This function belongs to `bap_add_polynom_COEFF.c`. It is run by 345 tests, including 20 in the `bap` library and 56 in the `DifferentialAlgebra` package.

`void bap_submulmon_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B, struct bav_term *TQ, COEFF_t q)` [Function]

Assign $A - q\,TQ\,B$ to R. This function belongs to `bap_add_polynom_COEFF.c`. It is run by 235 tests, including 1 in the `bap` library and 38 in the `DifferentialAlgebra` package.

`void bap_comblin_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, ba0_int_p cA, struct bap_polynom_COEFF *B, ba0_int_p cB)` [Function]

Assign the linear combination $cA\,A + cB\,B$ to R. This function belongs to `bap_add_polynom_COEFF.c`. It is run by 104 tests, including 1 in the `bap` library and 26 in the `DifferentialAlgebra` package.

`void bap_addmulrk_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_rank *rg, struct bap_polynom_COEFF *B)` [Function]

Assign $A\,rg + B$ to R. This function belongs to `bap_add_polynom_COEFF.c`. It is run by 7 tests, including 2 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`void bap_submulrk_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_rank *rg, struct bap_polynom_COEFF *B)` [Function]

Assign $A\,rg - B$ to R. This function belongs to `bap_add_polynom_COEFF.c`. It is run by 2 tests, including 2 in the `bap` library and 0 in the `DifferentialAlgebra` package.

6.3.5.3 Multiplication of polynomials

`void bap_neg_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A)` [Function]

Assign $-A$ to R. This function belongs to `bap_mul_polynom_COEFF.c`. It is run by 249 tests, including 13 in the `bap` library and 41 in the `DifferentialAlgebra` package.

`void bap_mul_polynom_numeric_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, COEFF_t c)` [Function]

Assign $c\,A$ to R. This function belongs to `bap_mul_polynom_COEFF.c`. It is run by 350 tests, including 19 in the `bap` library and 57 in the `DifferentialAlgebra` package.

`void bap_mul_polynom_value_int_p_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct ba0_value_int_p *val)` [Function]

Assign $A\,(x - \alpha)$ to R, where (x, α) denotes *val*. This function belongs to `bap_mul_polynom_COEFF.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_mul_polynom_variable_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_variable *v, bav_Idegree d)` [Function]

Assign $A\,v^d$ to R. This function belongs to `bap_mul_polynom_COEFF.c`. It is run by 43 tests, including 0 in the `bap` library and 12 in the `DifferentialAlgebra` package.

`void bap_mul_polynom_term_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_term *T)` [Function]

Assign $A\,T$ to R. This function belongs to `bap_mul_polynom_COEFF.c`. It is run by 337 tests, including 24 in the `bap` library and 56 in the `DifferentialAlgebra` package.

`void bap_mul_polynom_monom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, COEFF_t c, struct bav_term *T)` [Function]

Assign $A\,c\,T$ to R. This function belongs to `bap_mul_polynom_COEFF.c`. It is run by 336 tests, including 24 in the `bap` library and 56 in the `DifferentialAlgebra` package.

`void bap_mul_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)` [Function]

Assign $A\,B$ to R. This function belongs to `bap_mul_polynom_COEFF.c`. It is run by 362 tests, including 24 in the `bap` library and 59 in the `DifferentialAlgebra` package.

`void bap_pow_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, bav_Idegree n)` [Function]

Assign A^d to R. This function belongs to `bap_mul_polynom_COEFF.c`. It is run by 333 tests, including 23 in the `bap` library and 56 in the `DifferentialAlgebra` package.

6.3.5.4 Division and pseudo division of polynomials

`bool bap_is_numeric_factor_polynom_COEFF (struct bap_polynom_COEFF *A, COEFF_t c, struct bap_polynom_COEFF *Q)` [Function]

Return `true` if c divides A. If so and Q is not the zero pointer, it is assigned A/c. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 340 tests, including 2 in the `bap` library and 60 in the `DifferentialAlgebra` package.

`bool bap_is_variable_factor_polynom_COEFF (struct bap_polynom_COEFF *A, struct bav_variable *v, bav_Idegree *e)` [Function]

Return `true` if v divides A. If so and e is not the zero pointer, it is assigned the highest degree d such that v^d divides A. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 126 tests, including 0 in the `bap` library and 27 in the `DifferentialAlgebra` package.

`bool bap_is_factor_polynom_COEFF (struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B, struct bap_polynom_COEFF *Q)` [Function]

Return `true` if B divides A. If so and Q is not the zero pointer, it is assigned A/B. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 331 tests, including 3 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_exquo_polynom_term_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_term *T)` [Function]

Assign A/T to R. Assumes that the division is exact. May raise exception `BAV_EXEXQO` if division is not exact. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 331 tests, including 0 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_exquo_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)` [Function]

Assign A/B to R. Raise exception `BAV_EXEXQO` if the division is not exact. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 331 tests, including 3 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_exquo_polynom_product_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bap_product_COEFF *P)` [Function]

This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 206 tests, including 0 in the `bap` library and 35 in the `DifferentialAlgebra` package.

`void bap_pseudo_division_polynom_COEFF (struct bap_polynom_COEFF *Q, struct bap_polynom_COEFF *R, bav_Idegree *e, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B, struct bav_variable *x)` [Function]

Pseudo division of A by B, both polynomial being viewed as univariate polynomials in x, or in the leading variable of B, if x is zero. The pseudo quotient (resp. pseudo remainder) is stored in Q (resp. R) if this pointer is non zero. The integer $*e$ (if e is nonzero) is assigned the power to which the initial of B is raised to perform the division. If x is nonzero, it must be greater than or equal to the leader of B. If is is zero, B is not allowed to be numeric. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 101 tests, including 3 in the `bap` library and 10 in the `DifferentialAlgebra` package.

`void bap_prem_polynom_COEFF (struct bap_polynom_COEFF *R, bav_Idegree *e, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B, struct bav_variable *x)` [Function]

Variant of `bap_pseudo_division_polynom_COEFF`. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 77 tests, including 2 in the `bap` library and 9 in the `DifferentialAlgebra` package.

`void bap_pquo_polynom_COEFF (struct bap_polynom_COEFF *Q, bav_Idegree *e, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B, struct bav_variable *x)` [Function]

Variant of `bap_pseudo_division_polynom_COEFF`. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 33 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_rem_polynom_COEFF (struct bap_polynom_COEFF *Q,` [Function]
`struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A,`
`struct bap_polynom_COEFF *B, struct bav_variable *v)`

Division of A by B, both polynomial being viewed as univariate polynomials in x. It is assumed that the leading coefficient of B with respect to x divides exactly the leading coefficient of each remainder. If x is nonzero, it must be greater than or equal to the leader of B. If is is zero, B is not allowed to be numeric and x is understood to be the leading variable of B. This function belongs to `bap_prem_polynom_COEFF.c`. It is run by 1 tests, including 1 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_Euclidean_division_polynom_COEFF (struct` [Function]
`bap_polynom_COEFF *Q, struct bap_polynom_COEFF *R, struct`
`bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)`

This function belongs to `bap_invert_COEFF.c`. It is run by 136 tests, including 0 in the `bap` library and 23 in the `DifferentialAlgebra` package.

`void bap_Euclid_polynom_COEFF (struct bap_polynom_COEFF` [Function]
`*G, struct bap_polynom_COEFF *A, struct bap_polynom_COEFF *B)`

This function belongs to `bap_invert_COEFF.c`. It is run by 136 tests, including 0 in the `bap` library and 23 in the `DifferentialAlgebra` package.

`void bap_extended_Euclid_polynom_COEFF (struct` [Function]
`bap_polynom_COEFF *U, struct bap_polynom_COEFF *V, struct`
`bap_polynom_COEFF *G, struct bap_polynom_COEFF *A, struct`
`bap_polynom_COEFF *B)`

This function belongs to `bap_invert_COEFF.c`. It is run by 4 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

6.3.5.5 The subresultant algorithm

The following functions provide translations of the AXIOM functions of Lionel Ducos `prs.spad`. Functions suffixed by 2 compute single values. Functions suffixed by 3 apply to vectors.

`void bap_nsr2_Ducos_polynom_COEFF (struct` [Function]
`bap_polynom_COEFF *R, struct bap_polynom_COEFF *P, struct`
`bap_polynom_COEFF *Q, struct bap_polynom_COEFF *H, struct`
`bap_polynom_COEFF *s, struct bav_variable *v)`

Assign to R the next subresultant of P and Q with respect to v. The polynomial P has positive degree in v but Q and H do not necessarily. The variable v must be greater than or equal to all the variables occurring in the polynomials. Function `next_sousResultant2` of Lionel Ducos. This function belongs to `bap_Ducos_COEFF.c`. It is run by 7 tests, including 2 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`void bap_nsr3_Ducos_polynom_COEFF (struct` [Function]
`bap_tableof_polynom_COEFF *VR, struct`
`bap_tableof_polynom_COEFF *VP, struct`
`bap_tableof_polynom_COEFF *VQ, struct bap_polynom_COEFF *s,`
`struct bap_polynom_COEFF *ss, struct bav_variable *v)`

Function `next_sousResultant3` of Lionel Ducos. This function belongs to `bap_Ducos_COEFF.c`. It is run by 2 tests, including 2 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_muldiv_Lazard_polynom_COEFF (struct` [Function]
`bap_polynom_COEFF *R, struct bap_polynom_COEFF *x, struct`
`bap_polynom_COEFF *y, bav_Idegree n)`

Function `Lazard` of Lionel Ducos. Assign $x^n/y^{(n-1)}$ to R by alternating multiplications and divisions. This function belongs to `bap_Ducos_COEFF.c`. It is run by 2 tests, including 1 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_muldiv2_Lazard_polynom_COEFF (struct` [Function]
`bap_polynom_COEFF *R, struct bap_polynom_COEFF *F, struct`
`bap_polynom_COEFF *x, struct bap_polynom_COEFF *y,`
`bav_Idegree n)`

Function `Lazard2` of Lionel Ducos. Assign $F\,(x/y)^{(n-1)}$ to R. This function belongs to `bap_Ducos_COEFF.c`. It is run by 7 tests, including 2 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`void bap_muldiv3_Lazard_polynom_COEFF (struct` [Function]
`bap_tableof_polynom_COEFF *VR, struct`
`bap_tableof_polynom_COEFF *VP, struct bap_polynom_COEFF *x,`
`struct bap_polynom_COEFF *y, bav_Idegree n)`

Assign the array $x^{(n-1)}\,VP\,y^{(n-1)}$ to VR. This function belongs to `bap_Ducos_COEFF.c`. It is run by 3 tests, including 2 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_resultant2_Ducos_polynom_COEFF (struct` [Function]
`bap_product_COEFF *R, struct bap_polynom_COEFF *P, struct`
`bap_polynom_COEFF *Q, struct bav_variable *v)`

Assign to R the resultant with respect to v of P and Q. Function `resultant` of Lionel Ducos. This function belongs to `bap_Ducos_COEFF.c`. It is run by 10 tests, including 4 in the `bap` library and 1 in the `DifferentialAlgebra` package.

`void bap_lsr3_Ducos_polynom_COEFF (struct` [Function]
`bap_tableof_polynom_COEFF *VR, struct bap_polynom_COEFF *P,`
`struct bap_polynom_COEFF *Q, struct bav_variable *v)`

Variant of `lastSubResultantEuclidean` of Lionel Ducos. The array VR has size between 1 and 3. In the case the size is 3 assigns to VR the array (G, U, V) where G is the last nonzero subresultant of P and Q and where U and V satisfy $G = P\,U + Q\,V$. If the size is 2 only computes (G, U). If the size is 1 only computes (G). This function belongs to `bap_Ducos_COEFF.c`. It is run by 5 tests, including 4 in the `bap` library and 0 in the `DifferentialAlgebra` package.

6.3.5.6 Differentiation and related primitives

`bool bap_is_constant_polynom_COEFF (struct bap_polynom_COEFF *A, struct bav_symbol *s)` [Function]

Return `true` if the derivative of A with respect to s is zero, else `false`. If s is zero, then return `true` if the derivatives of A with respect to all independent variables are zero, else `false`. This function belongs to `bap_diff_polynom_COEFF.c`. It is run by 32 tests, including 0 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`bool bap_is_polynomial_with_constant_coefficients_COEFF (struct bap_polynom_COEFF *A, struct bav_variable *v, struct bav_symbol *x)` [Function]

Return `true` if A is a polynomial in v with coefficients constants with respect to derivation x. This function belongs to `bap_diff_polynom_COEFF.c`. It is run by 8 tests, including 0 in the `bap` library and 3 in the `DifferentialAlgebra` package.

`bool bap_is_independent_polynom_COEFF (struct bap_polynom_COEFF *A)` [Function]

Return `true` if A does not depend on dependent variables (i.e. does not depend on derivatives, unless they are parameters) else return `false`. This function belongs to `bap_diff_polynom_COEFF.c`. It is run by 233 tests, including 0 in the `bap` library and 36 in the `DifferentialAlgebra` package.

`void bap_diff_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_symbol *s)` [Function]

Assign to R the polynomial obtained by differentiating A with respect to s. Rewrite to zero any monomial involving a zero derivative of a parameter. This function belongs to `bap_diff_polynom_COEFF.c`. It is run by 190 tests, including 2 in the `bap` library and 42 in the `DifferentialAlgebra` package.

`void bap_diff2_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_term *T)` [Function]

Assign to R the polynomial obtained by differentiating A with respect to *theta*. Rewrite to zero any monomial involving a zero derivative of a parameter. The term *theta* encodes a derivation operator: every variable should correspond to a derivation. This function belongs to `bap_diff_polynom_COEFF.c`. It is run by 11 tests, including 1 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`void bap_simplify_zero_derivatives_of_parameter_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A)` [Function]

Simplify A by rewriting to zero the derivatives of the parameters present in P which turn out to be zero. Assign the result to R. This function belongs to `bap_parse_polynom_COEFF.c`. It is run by 4 tests, including 0 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`void bap_involved_parameters_polynom_COEFF (struct bav_dictionary_symbol *dict, struct bav_tableof_symbol *T, struct bap_polynom_COEFF *P)` [Function]

Append to T the parameters occurring in P. Every parameter already present in T is supposed to be registered in the dictionary *dict*. This function belongs to `bap_diff_polynom_COEFF.c`. It is run by 139 tests, including 0 in the `bap` library and 30 in the `DifferentialAlgebra` package.

6.3.5.7 Evaluation of polynomials

`void bap_set_point_polynom_COEFF (struct ba0_point_int_p *point, struct bap_polynom_COEFF *A, bool withld)` [Function]

Assign to *point* an evaluation point the variables of which being that of A and the values of which being zero. If *withld* is `false`, the leader of A is omitted in *point*. Variables occur in *point* by decreasing order. This function belongs to `bap_eval_polynom_COEFF.c`. It is run by 97 tests, including 0 in the `bap` library and 19 in the `DifferentialAlgebra` package.

`void bap_eval_to_polynom_at_numeric_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct bav_variable *v, COEFF_t c)` [Function]

Assign to R the polynomial obtained by evaluating A at $v = c$. This function belongs to `bap_eval_polynom_COEFF.c`. It is run by 59 tests, including 0 in the `bap` library and 4 in the `DifferentialAlgebra` package.

`void bap_eval_to_polynom_at_value_int_p_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct ba0_value_int_p *val)` [Function]

Assign to R the polynomial obtained by evaluating A at *val*. This function belongs to `bap_eval_polynom_COEFF.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_eval_to_polynom_at_point_int_p_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct ba0_point_int_p *point)` [Function]

Assign A mod *point* to R. This function belongs to `bap_eval_polynom_COEFF.c`. It is run by 36 tests, including 0 in the `bap` library and 9 in the `DifferentialAlgebra` package.

`void bap_eval_to_numeric_at_point_int_p_polynom_COEFF (COEFF_t *res, struct bap_polynom_COEFF *A, struct ba0_point_int_p *point)` [Function]

Assign A modulo *point* to *res*. Exception `BA0_ERRALG` is raised if some variables of A do not get evaluated. This function belongs to `bap_eval_polynom_COEFF.c`. It is run by 127 tests, including 0 in the `bap` library and 26 in the `DifferentialAlgebra` package.

`void bap_evalcoeff_at_point_int_p_polynom_COEFF (struct bap_polynom_COEFF *R, struct bap_polynom_COEFF *A, struct ba0_point_int_p *point)` [Function]

Assign to R the polynomial A modulo *point*. It is assumed that *point* gives values to all the variables of A apart its leader. This function belongs to `bap_eval_polynom_COEFF.c`. It is run by 97 tests, including 0 in the `bap` library and 19 in the `DifferentialAlgebra` package.

6.3.5.8 Geobuckets

Description of type `bap_geobucket_COEFF`. A geobucket (JSC (1998) 25, 285-293, Thomas Yan) is a table of polynomials representing the sum of these polynomials. This data structure permits to improve the complexity of the sum of many different polynomials. The polynomial at index i is a sum of at most $2\hat{}i$ monomials. The data structure is a duplicate of `bap_tableof_polynomial`. This type declaration belongs to `bap_geobucket_COEFF.h`.

```
struct bap_geobucket_COEFF
{
// number of allocated entries
  ba0_int_p alloc;
// number of used entries
  ba0_int_p size;
// the array
  struct bap_polynom_COEFF **tab;
};
```

`void bap_init_geobucket_COEFF (struct bap_geobucket_COEFF *geo)` [Function]

Initialize *geo* to 0. This function belongs to `bap_geobucket_COEFF.c`. It is run by 331 tests, including 30 in the `bap` library and 51 in the `DifferentialAlgebra` package.

`void bap_add_geobucket_COEFF (struct bap_geobucket_COEFF *geo, struct bap_polynom_COEFF *A)` [Function]

Add A to *geo*. This function belongs to `bap_geobucket_COEFF.c`. It is run by 328 tests, including 30 in the `bap` library and 49 in the `DifferentialAlgebra` package.

`void bap_sub_geobucket_COEFF (struct bap_geobucket_COEFF *geo, struct bap_polynom_COEFF *A)` [Function]

Subtract A from *geo*. This function belongs to `bap_geobucket_COEFF.c`. It is run by 241 tests, including 22 in the `bap` library and 37 in the `DifferentialAlgebra` package.

`void bap_reset_geobucket_COEFF (struct bap_geobucket_COEFF *geo)` [Function]

Reset *geo* to 0. This function belongs to `bap_geobucket_COEFF.c`. It is not run by any test.

`void bap_mul_geobucket_numeric_COEFF (struct bap_geobucket_COEFF *geo, COEFF_t c)` [Function]

Multiplie *geo* by *c*. This function belongs to `bap_geobucket_COEFF.c`. It is run by 1 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_mul_geobucket_COEFF (struct bap_geobucket_COEFF *geo, struct bap_polynom_COEFF *A)` [Function]

Multiplie *geo* by A. This function belongs to `bap_geobucket_COEFF.c`. It is run by 39 tests, including 0 in the `bap` library and 13 in the `DifferentialAlgebra` package.

`void bap_set_polynom_geobucket_COEFF (struct bap_polynom_COEFF *A, struct bap_geobucket_COEFF *geo)` [Function]

Assign *geo* to A. This function belongs to `bap_geobucket_COEFF.c`. It is run by 331 tests, including 30 in the `bap` library and 51 in the `DifferentialAlgebra` package.

6.3.6 Polynomials with `mpz_t` coefficients

6.3.6.1 Primitives applying on numerical coefficients

`void bap_polynom_mpq_to_mpz (struct bap_polynom_mpz *A, struct bap_polynom_mpq *B)` [Function]

Assign the numerator of B to A. This function belongs to `bap_polyspec_mpz.c`. It is not run by any test.

`void bap_normal_sign_polynom_mpz (struct bap_polynom_mpz *A, struct bap_polynom_mpz *B)` [Function]

Assign to \$pm B$ so that the leading coefficient of A is positive. This function belongs to `bap_polyspec_mpz.c`. It is not run by any test.

`void bap_maxnorm_polynom_mpz (ba0_mpz_t n, struct bap_polynom_mpz *A)` [Function]

Assign to n the maximum of the absolute values of the coefficients of A. This function belongs to `bap_polyspec_mpz.c`. It is run by 105 tests, including 0 in the `bap` library and 13 in the `DifferentialAlgebra` package.

`void bap_numeric_content_polynom_mpz (ba0_mpz_t n, struct bap_polynom_mpz *A)` [Function]

Assign to n the numerical content of A. This function belongs to `bap_polyspec_mpz.c`. It is run by 299 tests, including 0 in the `bap` library and 59 in the `DifferentialAlgebra` package.

`void bap_signed_numeric_content_polynom_mpz (ba0_mpz_t n, struct bap_polynom_mpz *A)` [Function]

Assign to n the numerical content of A. The content carries the same sign as the leading coefficient of A. This function belongs to `bap_polyspec_mpz.c`. It is run by 331 tests, including 0 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_numeric_primpart_polynom_mpz (struct bap_polynom_mpz *A, struct bap_polynom_mpz *B)` [Function]

Assign to A the numerical primitive part of B. This function belongs to `bap_polyspec_mpz.c`. It is run by 117 tests, including 0 in the `bap` library and 21 in the `DifferentialAlgebra` package.

`void bap_normal_numeric_primpart_polynom_mpz (struct bap_polynom_mpz *A, struct bap_polynom_mpz *B)` [Function]

Assign to A the numerical primitive part of B. The leading coefficient of A is positive. This function belongs to `bap_polyspec_mpz.c`. It is run by 112 tests, including 0 in the `bap` library and 14 in the `DifferentialAlgebra` package.

`bool bap_is_numeric_factor_polynom_mpz (struct bap_polynom_mpz *A, ba0_mpz_t n, struct bap_polynom_mpz *R)` [Function]

Return `true` if n divides A. If so and R is not the zero pointer then it is assigned A/n. This function belongs to `bap_polyspec_mpz.c`. It is run by 340 tests, including 2 in the `bap` library and 60 in the `DifferentialAlgebra` package.

`void bap_exquo_polynom_numeric_mpz (struct bap_polynom_mpz *R, struct bap_polynom_mpz *A, ba0_mpz_t n)` [Function]

Assign to R the polynomial A/n. The division is assumed to be exact. May raise exception `BAV_EXEXQO`. This function belongs to `bap_polyspec_mpz.c`. It is run by 338 tests, including 0 in the `bap` library and 60 in the `DifferentialAlgebra` package.

6.3.7 Polynomials with `mpq_t` coefficients

`void bap_polynom_mpz_to_mpq (struct bap_polynom_mpq *A, struct bap_polynom_mpz *B)` [Function]

Assign B to A. This function belongs to `bap_polyspec_mpq.c`. It is run by 46 tests, including 0 in the `bap` library and 9 in the `DifferentialAlgebra` package.

`void bap_set_polynom_numer_denom_mpq (struct bap_polynom_mpq *A, struct bap_polynom_mpz *B, ba0_mpz_t denom)` [Function]

Assign $B/denom$ to A. If *denom* is the zero pointer then it is supposed to be equal to 1. Exception `BA0_ERRIVZ` is raised if the value of *denom* is zero. This function belongs to `bap_polyspec_mpq.c`. It is run by 49 tests, including 0 in the `bap` library and 9 in the `DifferentialAlgebra` package.

`void bap_numer_polynom_mpq (struct bap_polynom_mpz *A, ba0_mpz_t denom, struct bap_polynom_mpq *B)` [Function]

Assign to A and *denom* the numerator and the denominator of B. The parameter *denom* may be the zero pointer. This function belongs to `bap_polyspec_mpq.c`. It is run by 81 tests, including 0 in the `bap` library and 16 in the `DifferentialAlgebra` package.

`void bap_denom_polynom_mpq (ba0_mpz_t n, struct bap_polynom_mpq *A)` [Function]

Assign to *denom* the denominator of B. This function belongs to `bap_polyspec_mpq.c`. It is run by 81 tests, including 0 in the `bap` library and 16 in the `DifferentialAlgebra` package.

6.3.8 Polynomials with modular mpz_t coefficients

`void bap_polynom_mpz_to_mpzm (struct bap_polynom_mpzm *R, struct bap_polynom_mpz *A)` [Function]

Assign to R the polynomial A modulo `ba0_mpzm_module`. This function belongs to `bap_polyspec_mpzm.c`. It is run by 16 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_polynom_mint_hp_to_mpzm (struct bap_polynom_mpzm *R, struct bap_polynom_mint_hp *A)` [Function]

Assign to R the polynomial A, assuming that `ba0_mpzm_module` is a multiple of `ba0_mint_hp_module`. This function belongs to `bap_polyspec_mpzm.c`. It is run by 112 tests, including 0 in the `bap` library and 21 in the `DifferentialAlgebra` package.

`void bap_mods_polynom_mpzm (struct bap_polynom_mpz *R, struct bap_polynom_mpzm *A)` [Function]

Assign to R the polynomial A. Maps the coefficients to the signed integers with absolute value less than one half of `ba0_mpzm_module`. This function belongs to `bap_polyspec_mpzm.c`. It is run by 112 tests, including 0 in the `bap` library and 21 in the `DifferentialAlgebra` package.

`void bap_mods_product_mpzm (struct bap_product_mpz *P, struct bap_product_mpzm *Q)` [Function]

Variant of the above function, for products. This function belongs to `bap_polyspec_mpzm.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_Bezout_polynom_mpzm (struct bap_polynom_mpzm *U, struct bap_polynom_mpzm *V, struct bap_polynom_mpzm *G, struct bap_polynom_mpzm *A, struct bap_polynom_mpzm *B, ba0_mpz_t p, bav_Idegree k)` [Function]

Assign to U, V and G polynomials such that G is the gcd of A and B and $AU + BV = G$ modulo p^k. The global variable `ba0_mpzm_module` is changed. Exception `BAP_EXHNCP` is raised if the computation failed. This function belongs to `bap_polyspec_mpzm.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_coeftayl_polynom_mpzm (struct bap_polynom_mpzm *R, struct bap_polynom_mpzm *A, struct ba0_value_int_p *val, bav_Idegree k)` [Function]

Assign to R the coefficient of $(x - \alpha)$ in the Taylor expansion of A at $val = (x, \alpha)$. This function belongs to `bap_polyspec_mpzm.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_quorem_polynom_mpzm (struct bap_polynom_mpzm *Q, struct bap_polynom_mpzm *R, struct bav_rank *rg, struct bap_polynom_mpzm *A, struct bap_polynom_mpzm *B)` [Function]

Assign to R and Q polynomials such that $rg\,A = B\,Q + R$ with $\deg R < \deg B$. Assumes $\deg A < \deg B$. This function belongs to `bap_polyspec_mpzm.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_uni_Diophante_polynom_mpzm (struct bap_polynom_mpzm *sigma, struct bap_product_mpzm *prod, struct bav_rank *rg, ba0_mpz_t p, bav_Idegree k)` [Function]

Denote $prod = a_1 \cdot a_r$ and $b_i = a_1 \cdots a_{i-1}\, a_{i+1} \cdots a_r$. Assigns to *sigma* the unique solution of the Diophantine polynomial equation in $(Z/p^k Z)[x]$ $\sigma_1\, b_1 + \cdots + \sigma_r\, b_r = rg$ modulo p^k such that $\deg \sigma_i < \deg a_i$. Assumes `ba0_mpzm_module` is p^k, that p does not divide any initial of the a_i and that the a_i are relatively prime modulo p. This function belongs to `bap_polyspec_mpzm.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_multi_Diophante_polynom_mpzm (struct bap_polynom_mpzm *sigma, struct bap_product_mpzm *prod, struct bap_polynom_mpzm *C, struct ba0_point_int_p *point, bav_Idegree maxdeg, ba0_mpz_t p, bav_Idegree k)` [Function]

Denote $prod = a_1 \cdot a_r$ and $b_i = a_1 \cdots a_{i-1}\, a_{i+1} \cdots a_r$. Assigns to *sigma* the unique solution in $Z[x, y_1, \ldots, y_t]$ of the Diophantine polynomial equation $\sigma_1\, b_1 + \cdots + \sigma_r\, b_r = C$ modulo $(point^{maxdeg}, p^k)$ such that $\deg(\sigma_i, x) < \deg(a_i, x)$. Assumes that `ba0_mpzm_module` is p^k, that p does not divide any initial of the a_i modulo *point* and that the a_i are relatively prime modulo $(point, p^k)$. This function belongs to `bap_polyspec_mpzm.c`. It is run by 2 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

6.3.9 Polynomials with modular machine integer coefficients

`void bap_polynom_mpz_to_mint_hp (struct bap_polynom_mint_hp *R, struct bap_polynom_mpz *A)` [Function]

Assign to R the polynomial A modulo `ba0_mint_hp_module`. This function belongs to `bap_polyspec_mint_hp.c`. It is run by 136 tests, including 0 in the `bap` library and 23 in the `DifferentialAlgebra` package.

`void bap_random_eval_polynom_mpz_to_mint_hp (struct bap_polynom_mint_hp *R, struct bap_polynom_mpz *A, ba0_unary_predicate *f)` [Function]

Assign to R the polynomial A modulo $(p, x_1 - \alpha_1, \ldots, x_k - \alpha_k)$ where p denotes `ba0_mint_hp_module`, the x_k denote the derivatives occurring in A which satisfy f and the $\alpha_k = g(p, x_k)$ are "random" values which are actually functions of p and x_k. This function belongs to `bap_polyspec_mint_hp.c`. It is run by 59 tests, including 0 in the `bap` library and 8 in the `DifferentialAlgebra` package.

`void bap_Berlekamp_mint_hp (struct bap_product_mint_hp *prod, struct bap_polynom_mint_hp *P)` [Function]

Denote n the module `ba0_mint_hp_module`. Assigns to *prod* the factorization $1/i_P\, P = F_1 \cdots F_r$ modulo n where the F_i are irreducible modulo n. The input polynomial P is assumed to be nonconstant and squarefree. This function belongs to `bap_polyspec_mint_hp.c`. It is run by 3 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

6.4 Iterators and creators for polynomials

Two types of iterators are provided (iterators of monomials and iterators of coefficients w.r.t. some variable). A creator is also provided to build polynomials monomial per monomial.

6.4.1 Enumerating polynomial coefficients

`void bap_begin_itercoeff_COEFF (struct bap_itercoeff_COEFF *I, struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Initialize the iterator on the first coefficient of A i.e. the one with highest term, with respect to the lexicographic order induced by the current ordering. The variable v provides the lowest variable of the alphabet of the terms. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 303 tests, including 12 in the `bap` library and 42 in the `DifferentialAlgebra` package.

`void bap_end_itercoeff_COEFF (struct bap_itercoeff_COEFF *I, struct bap_polynom_COEFF *A, struct bav_variable *v)` [Function]

Initialize the iterator on the last coefficient of A i.e. the one with lowest term, with respect to the lexicographic order induced by the current ordering. The variable v provides the lowest variable of the alphabet of the terms. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 113 tests, including 2 in the `bap` library and 11 in the `DifferentialAlgebra` package.

`bool bap_outof_itercoeff_COEFF (struct bap_itercoeff_COEFF *I)` [Function]

Return true if the iterator has exited from the polynomial. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 314 tests, including 11 in the `bap` library and 43 in the `DifferentialAlgebra` package.

`void bap_next_itercoeff_COEFF (struct bap_itercoeff_COEFF *I)` [Function]

Move the iterator to the next term. Exception `BAO_ERRALG` is raised if the iterator is outside the polynomial before the call. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 271 tests, including 7 in the `bap` library and 38 in the `DifferentialAlgebra` package.

`void bap_prev_itercoeff_COEFF (struct bap_itercoeff_COEFF *I)` [Function]

Move the iterator to the previous term. Exception `BAO_ERRALG` is raised if the iterator is outside the polynomial before the call. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 113 tests, including 2 in the `bap` library and 11 in the `DifferentialAlgebra` package.

`void bap_term_itercoeff_COEFF (struct bav_term *T, struct bap_itercoeff_COEFF *I)` [Function]

Assign to T the current term. Exception `BAO_ERRALG` is raised if the iterator is outside the polynomial before the call. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 253 tests, including 5 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_coeff_itercoeff_COEFF (struct bap_polynom_COEFF *A, struct bap_itercoeff_COEFF *I)` [Function]

Assign to A the current coefficient in readonly mode. Exception `BAO_ERRALG` is raised if the iterator is outside the polynomial before the call. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 314 tests, including 11 in the `bap` library and 43 in the `DifferentialAlgebra` package.

`void bap_seek_coeff_itercoeff_COEFF (struct bap_polynom_COEFF *A, struct bap_itercoeff_COEFF *I, struct bav_term *T)` [Function]

Assign to A, as a readonly polynomial, the coefficient of the term T. If T involves variables strictly less than the last variable of the alphabet of the terms, they are ignored. Exception `BAO_ERRALG` is raised if the iterator is outside the polynomial before the call. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 164 tests, including 3 in the `bap` library and 20 in the `DifferentialAlgebra` package.

`void bap_close_itercoeff_COEFF (struct bap_itercoeff_COEFF *iter)` [Function]

Close the iterator. This function belongs to `bap_itercoeff_COEFF.c`. It is run by 284 tests, including 4 in the `bap` library and 38 in the `DifferentialAlgebra` package.

6.4.2 Enumerating and creating numerical coefficients

6.4.2.1 Enumerating numerical coefficients

Description of type `bap_itermon_COEFF`. This data type implements an iterator of monomials of a differential polynomials, viewed as multivariate polynomial over the numerical coefficients. The monomials of the polynomial may be accessed either in sequential access as in indexed access. This type declaration belongs to `bap_itermon_COEFF.h`.

```
struct bap_itermon_COEFF
{
  struct bap_polynom_COEFF *poly;       // the polynomial
// an iterator of monomials of poly->clot
  struct bap_itermon_clot_COEFF iter;
// an auxiliary data structure if access is indexed
// it provides the current index in poly->ind.tab
  struct bap_iterator_indexed_access iter_ix;
};
```

`void bap_begin_itermon_COEFF (struct bap_itermon_COEFF *I, struct bap_polynom_COEFF *A)` [Function]

Initialize the iterator I and set it on the first monomial of A. This function belongs to `bap_itermon_COEFF.c`. It is run by 383 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_end_itermon_COEFF (struct bap_itermon_COEFF *I, struct bap_polynom_COEFF *A)` [Function]

Initialize the iterator I and set it on the last monomial of A. This function belongs to `bap_itermon_COEFF.c`. It is run by 334 tests, including 3 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_set_itermon_COEFF (struct bap_itermon_COEFF *I, struct bap_itermon_COEFF *J)` [Function]

Assign J to I. This function belongs to `bap_itermon_COEFF.c`. It is run by 303 tests, including 10 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`bool bap_outof_itermon_COEFF (struct bap_itermon_COEFF *I)` [Function]

Return true if the iterator is outside the polynomial. This function belongs to `bap_itermon_COEFF.c`. It is run by 382 tests, including 31 in the `bap` library and 63 in the `DifferentialAlgebra` package.

`void bap_next_itermon_COEFF (struct bap_itermon_COEFF *I)` [Function]

Move I to the next monomial. This function belongs to `bap_itermon_COEFF.c`. It is run by 383 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_prev_itermon_COEFF (struct bap_itermon_COEFF *I)` [Function]

Move I to the previous monomial. This function belongs to `bap_itermon_COEFF.c`. It is run by 198 tests, including 3 in the `bap` library and 27 in the `DifferentialAlgebra` package.

`void bap_goto_itermon_COEFF (struct bap_itermon_COEFF *I, ba0_int_p n)` [Function]

Move I on the monomial with number n. Exception `BA0_ERRALG` is raised if n is not a valid monomial number. This function belongs to `bap_itermon_COEFF.c`. It is run by 253 tests, including 10 in the `bap` library and 40 in the `DifferentialAlgebra` package.

`COEFF_t * bap_coeff_itermon_COEFF (struct bap_itermon_COEFF *I)` [Function]

Return the address of the coefficient of the current monomial. This function belongs to `bap_itermon_COEFF.c`. It is run by 383 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_term_itermon_COEFF (struct bav_term *T, struct bap_itermon_COEFF *I)` [Function]

Assign to T the term of the current monomial. This function belongs to `bap_itermon_COEFF.c`. It is run by 382 tests, including 31 in the `bap` library and 63 in the `DifferentialAlgebra` package.

`void bap_reductum_itermon_COEFF (struct bap_itermon_COEFF *I, struct bap_polynom_COEFF *A)` [Function]

Assign to A the sum of all the monomials following the current monomial (this one included). The result is readonly. This function belongs to `bap_itermon_COEFF.c`. It is run by 222 tests, including 4 in the `bap` library and 35 in the `DifferentialAlgebra` package.

Description of type `bap_rank_code`. This data type is used as return code for comparing ranks. This type declaration belongs to `bap_common.h`.

```
enum bap_rank_code
{
```

```
  bap_rank_too_low,
  bap_rank_ok,
  bap_rank_too_high
};
```

`void bap_seekfirst_itermon_COEFF (struct bap_itermon_COEFF *I, enum bap_rank_code (*f) (struct bav_term *, struct bav_term *, bav_Inumber), struct bav_term *last_term, bav_Inumber last_number)` [Function]

Move the iterator I to the first monomial such that $f(T)$ returns `bap_rank_ok`, where T denotes the term of the monomial. Arguments *last_term* and *last_number* are extra arguments passed to *f*. The monomial is obtained by dichotomic search. This function belongs to `bap_itermon_COEFF.c`. It is run by 231 tests, including 5 in the `bap` library and 22 in the `DifferentialAlgebra` package.

`void bap_seeklast_itermon_COEFF (struct bap_itermon_COEFF *I, enum bap_rank_code (*f) (struct bav_term *, struct bav_term *, bav_Inumber), struct bav_term *last_term, bav_Inumber last_number)` [Function]

Move the iterator I to the last monomial such that $f(T)$ returns `bap_rank_ok`, where T denotes the term of the monomial. Arguments *last_term* and *last_number* are extra arguments passed to *f*. The monomial is obtained by dichotomic search. This function belongs to `bap_itermon_COEFF.c`. It is run by 303 tests, including 12 in the `bap` library and 42 in the `DifferentialAlgebra` package.

`void bap_close_itermon_COEFF (struct bap_itermon_COEFF *iter)` [Function]

Close the iterator. This function belongs to `bap_itermon_COEFF.c`. It is run by 107 tests, including 1 in the `bap` library and 26 in the `DifferentialAlgebra` package.

6.4.2.2 Creating polynomials

Description of type `bap_typeof_total_rank`. This data type provides information on a `total_rank` field while creating a polynomial, monomial per monomial. This type declaration belongs to `bap_common.h`.

```
enum bap_typeof_total_rank
{
  bap_exact_total_rank,
  bap_approx_total_rank
};
```

Description of type `bap_creator_COEFF`. This data type permits to create a polynomial from scratch by giving, in decreasing order, its terms and numerical coefficients. This type declaration belongs to `bap_creator_COEFF.h`.

```
struct bap_creator_COEFF
{
// the polynomial to create
  struct bap_polynom_COEFF *poly;
// a creator for the clot of the polynomial
  struct bap_creator_clot_COEFF crea;
// the type of the total rank provided when initializing the process
  enum bap_typeof_total_rank type;
};
```

`void bap_begin_creator_COEFF (struct bap_creator_COEFF *crea, struct bap_polynom_COEFF *A, struct bav_term *T, enum bap_typeof_total_rank type, ba0_int_p table2of_monom_alloc)` [Function]

Initialize the creator *crea* at the beginning of A. The polynomial A is going to be overwritten. If allocating some `bap_mont_COEFF` in the clot of A is necessary they will be allocated *nbmon_par_mont* monomials. The parameter *type* indicates if T is the exact total rank of the resulting polynomial or a mere upper bound. This function belongs to `bap_creator_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_append_creator_COEFF (struct bap_creator_COEFF *crea, struct bap_polynom_COEFF *A, ba0_int_p table2of_monom_alloc)` [Function]

Initialize the creator *crea* at the end of the polynomial A (further monomials are going to be appended to the ones of A). If allocating some `bap_mont_COEFF` in the clot of A is necessary they will be allocated *nbmon_par_mont* monomials. This function belongs to `bap_creator_COEFF.c`. It is run by 4 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

`void bap_write_creator_COEFF (struct bap_creator_COEFF *crea, struct bav_term *T, COEFF_t c)` [Function]

Write cT on the creator *crea*. The coefficient may be zero. This function belongs to `bap_creator_COEFF.c`. It is run by 384 tests, including 31 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_write_neg_creator_COEFF (struct bap_creator_COEFF *crea, struct bav_term *T, COEFF_t c)` [Function]

Write $-cT$ on the creator. The coefficient may be zero. This function belongs to `bap_creator_COEFF.c`. It is run by 330 tests, including 18 in the `bap` library and 48 in the `DifferentialAlgebra` package.

`bool bap_is_write_allable_creator_COEFF (struct bap_creator_COEFF *crea, struct bap_polynom_COEFF *A)` [Function]

Return `true` if the function `bap_write_all_creator_COEFF` can be applied to *crea* and A. This function belongs to `bap_creator_COEFF.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_write_all_creator_COEFF (struct bap_creator_COEFF *crea, struct bap_polynom_COEFF *A)` [Function]

Write all monomials of A on the creator *crea* efficiently. This function only applies in restricted cases. In particular, it is required that the term managers of A and of the `poly` field of *crea* are compatible. This function belongs to `bap_creator_COEFF.c`. It is run by 383 tests, including 30 in the `bap` library and 64 in the `DifferentialAlgebra` package.

`void bap_write_neg_all_creator_COEFF (struct bap_creator_COEFF *crea, struct bap_polynom_COEFF *A)` [Function]

Write all monomials of $-A$ on the creator *crea* efficiently. This function only applies in restricted cases. In particular, it is required that the term managers of A and of the

poly field of *crea* are compatible. This function belongs to bap_creator_COEFF.c. It is run by 112 tests, including 13 in the bap library and 27 in the DifferentialAlgebra package.

`void bap_write_mul_all_creator_COEFF (struct bap_creator_COEFF *crea, struct bap_polynom_COEFF *A, COEFF_t c)` [Function]

Write all monomials of $c\,A$ on the creator *crea* efficiently. This function only applies in restricted cases. In particular, it is required that the term managers of A and of the poly field of *crea* are compatible. This function belongs to bap_creator_COEFF.c. It is run by 259 tests, including 16 in the bap library and 35 in the DifferentialAlgebra package.

`void bap_write_exquo_all_creator_mpz (struct bap_creator_mpz *crea, struct bap_polynom_mpz *A, ba0_mpz_t c)` [Function]

This function is only available for polynomials with ba0_mpz_t coefficients. Write all monomials of A/c on the creator *crea* efficiently. The division of the coefficients of A by c must be exact. This function only applies in restricted cases. In particular, it is required that the term managers of A and of the poly field of *crea* are compatible. This function belongs to bap_creator_COEFF.c. It is not run by any test.

`void bap_close_creator_COEFF (struct bap_creator_COEFF *crea)` [Function]

Complete the construction of the poly field of *crea*. If type is bap_approx_total_rank then the total_rank field of A is recomputed. This function belongs to bap_creator_COEFF.c. It is run by 384 tests, including 31 in the bap library and 64 in the DifferentialAlgebra package.

6.5 Products

Description of type bap_power_COEFF. This data type implements a polynomial raised to some power. This type declaration belongs to bap_product_COEFF.h.

```
struct bap_power_COEFF
{
  struct bap_polynom_COEFF factor;
  bav_Idegree exponent;
};
```

Description of type bap_product_COEFF. This data type implements products of polynomials of the form $c\,f_1^{a_1}\cdots f_t^{a_t}$ where c is the *numerical factor* of the product, the f_i are non numerical polynomials, not necessarily irreducible and not even necessarily pairwise distinct. The exponents a_i are allowed to be zero. The product one is coded by $c = 1$ and $t = 0$. The product zero is coded by $c = 0$. Observe that in the context of rings which are not domains, zero and one are not necessarily automatically recognized. This type declaration belongs to bap_product_COEFF.h.

```
struct bap_product_COEFF
{
// the numerical factor
  COEFF_t num_factor;
```

```
// the number of allocated entries of tab
  ba0_int_p alloc;
// the number of used entries
  ba0_int_p size;
// the array of bap_power_COEFF
  struct bap_power_COEFF *tab;
};
```

The format identifiers for products are `%Pz` (products with `mpz_t` coefficients), `%Pq` (products with rational numbers coefficients), `%Pzm` (products with modular `mpz_t` coefficients), and `%Pim` (products with modular machine integers coefficients).

`void bap_init_power_COEFF (struct bap_power_COEFF *P)` [Function]
Initialize P to 1[1]. This function belongs to `bap_product_COEFF.c`. It is run by 375 tests, including 30 in the `bap` library and 57 in the `DifferentialAlgebra` package.

`struct bap_power_COEFF * bap_new_power_COEFF (void)` [Function]
Allocate a new power, initialize it and return it. This function belongs to `bap_product_COEFF.c`. It is run by 280 tests, including 30 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_set_power_COEFF (struct bap_power_COEFF *P, struct bap_power_COEFF *Q)` [Function]
Assign Q to P. This function belongs to `bap_product_COEFF.c`. It is run by 272 tests, including 26 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_set_power_polynom_COEFF (struct bap_power_COEFF *P, struct bap_polynom_COEFF *A, bav_Idegree k)` [Function]
Assign A^k to P. This function belongs to `bap_product_COEFF.c`. It is not run by any test.

`void bap_pow_power_COEFF (struct bap_power_COEFF *P, struct bap_power_COEFF *Q, bav_Idegree k)` [Function]
Assign Q^k to P. This function belongs to `bap_product_COEFF.c`. It is not run by any test.

6.5.1 Generic products

`void bap_init_product_COEFF (struct bap_product_COEFF *prod)` [Function]
Initialize *prod* to one. This function belongs to `bap_product_COEFF.c`. It is run by 375 tests, including 30 in the `bap` library and 57 in the `DifferentialAlgebra` package.

`void bap_init_product_zero_COEFF (struct bap_product_COEFF *prod)` [Function]
Initialize *prod* to zero. This function belongs to `bap_product_COEFF.c`. It is not run by any test.

`void bap_realloc_product_COEFF (struct bap_product_COEFF *prod, ba0_int_p n)` [Function]
Reallocate *prod* if needed so that it can receive a product of at least n powers. The already existing powers are kept. This function belongs to `bap_product_COEFF.c`. It

is run by 375 tests, including 30 in the `bap` library and 57 in the `DifferentialAlgebra` package.

`struct bap_product_COEFF * bap_new_product_COEFF (void)` [Function]
Allocate a new product, initialize it and return it. This function belongs to `bap_product_COEFF.c`. It is run by 312 tests, including 30 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`struct bap_product_COEFF * bap_new_product_zero_COEFF (void)` [Function]
Return the product zero. This function belongs to `bap_product_COEFF.c`. It is not run by any test.

`void bap_set_product_one_COEFF (struct bap_product_COEFF *P)` [Function]
Assign 1 to P. This function belongs to `bap_product_COEFF.c`. It is run by 338 tests, including 2 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_set_product_zero_COEFF (struct bap_product_COEFF *P)` [Function]
Assign 0 to P. This function belongs to `bap_product_COEFF.c`. It is run by 204 tests, including 11 in the `bap` library and 33 in the `DifferentialAlgebra` package.

`void bap_set_product_numeric_COEFF (struct bap_product_COEFF *P, COEFF_t c)` [Function]
Assign c to P. This function belongs to `bap_product_COEFF.c`. It is run by 215 tests, including 2 in the `bap` library and 33 in the `DifferentialAlgebra` package.

`void bap_set_product_polynom_COEFF (struct bap_product_COEFF *P, struct bap_polynom_COEFF *A, bav_Idegree d)` [Function]
Assign A^d to P. This function belongs to `bap_product_COEFF.c`. It is run by 225 tests, including 4 in the `bap` library and 33 in the `DifferentialAlgebra` package.

`bool bap_is_one_product_COEFF (struct bap_product_COEFF *P)` [Function]
Return `true` if P is equal to 1. This function belongs to `bap_product_COEFF.c`. It is run by 118 tests, including 0 in the `bap` library and 27 in the `DifferentialAlgebra` package.

`bool bap_is_zero_product_COEFF (struct bap_product_COEFF *P)` [Function]
Return `true` if P is equel to 0. This function belongs to `bap_product_COEFF.c`. It is run by 374 tests, including 30 in the `bap` library and 57 in the `DifferentialAlgebra` package.

`bool bap_is_numeric_product_COEFF (struct bap_product_COEFF *P)` [Function]
Return `true` if P is a numerical coefficient (the function actually only tests if all exponents are equal to 0). This function belongs to `bap_product_COEFF.c`. It is run by 201 tests, including 0 in the `bap` library and 33 in the `DifferentialAlgebra` package.

`struct bav_variable * bap_leader_product_COEFF (struct bap_product_COEFF *P)` [Function]

Return the maximum of the leaders of the factors of P. Exception `BAP_ERRCST` is raised if P is numeric. This function belongs to `bap_product_COEFF.c`. It is not run by any test.

`void bap_set_product_COEFF (struct bap_product_COEFF *P, struct bap_product_COEFF *Q)` [Function]

Assign Q to P. Do not copy the powers whose exponents are zero. This function belongs to `bap_product_COEFF.c`. It is run by 331 tests, including 0 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`bav_Idegree bap_exponent_product_COEFF (struct bap_product_COEFF *prod, struct bap_polynom_COEFF *A)` [Function]

If A is one of the factors of *prod*, return its exponent. Return 0 otherwise. This function belongs to `bap_product_COEFF.c`. It is not run by any test.

`void bap_sort_product_COEFF (struct bap_product_COEFF *P, struct bap_product_COEFF *Q)` [Function]

Apply `bap_sort_polynom_COEFF` over all the factors of Q. The result is stored in P in readonly mode. This function belongs to `bap_product_COEFF.c`. It is run by 98 tests, including 0 in the `bap` library and 24 in the `DifferentialAlgebra` package.

`void bap_physort_product_COEFF (struct bap_product_COEFF *P)` [Function]

Apply `bap_physort_polynom_COEFF` over all the factors of P. This function belongs to `bap_product_COEFF.c`. It is run by 18 tests, including 0 in the `bap` library and 3 in the `DifferentialAlgebra` package.

`void bap_expand_product_COEFF (struct bap_polynom_COEFF *R, struct bap_product_COEFF *prod)` [Function]

Assign to R the expansion of *prod*. This function belongs to `bap_product_COEFF.c`. It is run by 374 tests, including 30 in the `bap` library and 57 in the `DifferentialAlgebra` package.

`void bap_mul_product_polynom_COEFF (struct bap_product_COEFF *res, struct bap_product_COEFF *prod, struct bap_polynom_COEFF *A, bav_Idegree k)` [Function]

Assign to *res* the product *prod* A^k. This function belongs to `bap_product_COEFF.c`. It is run by 256 tests, including 0 in the `bap` library and 35 in the `DifferentialAlgebra` package.

`void bap_neg_product_COEFF (struct bap_product_COEFF *R, struct bap_product_COEFF *A)` [Function]

Assign to R the product $-A$. This function belongs to `bap_product_COEFF.c`. It is not run by any test.

`void bap_mul_product_numeric_COEFF (struct bap_product_COEFF *R, struct bap_product_COEFF *A, COEFF_t c)` [Function]

Assign to R the product $A\,c$. This function belongs to `bap_product_COEFF.c`. It is run by 331 tests, including 0 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_mul_product_term_COEFF (struct bap_product_COEFF *res, struct bap_product_COEFF *prod, struct bav_term *T)` [Function]

Assign to R the product $prod\,T$. This function belongs to `bap_product_COEFF.c`. It is run by 331 tests, including 0 in the `bap` library and 53 in the `DifferentialAlgebra` package.

`void bap_mul_product_COEFF (struct bap_product_COEFF *R, struct bap_product_COEFF *P, struct bap_product_COEFF *Q)` [Function]

Assign to R the product $P\,Q$. This function belongs to `bap_product_COEFF.c`. It is run by 237 tests, including 0 in the `bap` library and 35 in the `DifferentialAlgebra` package.

`void bap_pow_product_COEFF (struct bap_product_COEFF *P, struct bap_product_COEFF *Q, bav_Idegree k)` [Function]

Assign to P the product Q^k. This function belongs to `bap_product_COEFF.c`. It is run by 218 tests, including 0 in the `bap` library and 33 in the `DifferentialAlgebra` package.

`void bap_exquo_product_polynom_COEFF (struct bap_product_COEFF *res, struct bap_product_COEFF *prod, struct bap_polynom_COEFF *A, bav_Idegree k)` [Function]

Assign to *res* the product $prod/A^k$. The polynomial A must be one of the factors of the product and the division must be exact. May raise exception `BAV_EXEXQO`. This function belongs to `bap_product_COEFF.c`. It is run by 8 tests, including 0 in the `bap` library and 2 in the `DifferentialAlgebra` package.

`void * bap_scanf_product_COEFF (void *A)` [Function]

The general parsing function for products. It is called by `ba0_scanf/%P_`. This function belongs to `bap_product_COEFF.c`. It is run by 280 tests, including 30 in the `bap` library and 39 in the `DifferentialAlgebra` package.

`void bap_printf_product_COEFF (void *A)` [Function]

The general printing function for products. It is called by `ba0_printf/%P_`. This function belongs to `bap_product_COEFF.c`. It is run by 38 tests, including 0 in the `bap` library and 5 in the `DifferentialAlgebra` package.

6.5.2 Products with `mpq_t` coefficients

`void bap_product_mpz_to_mpq (struct bap_product_mpq *R, struct bap_product_mpz *P)` [Function]

Assign B to A. This function belongs to `bap_polyspec_mpq.c`. It is run by 1 tests, including 0 in the `bap` library and 0 in the `DifferentialAlgebra` package.

7 The baz library

7.1 Introduction

The baz library implements important algorithms for differential polynomials over the GMP big integers (mpz_t, ba0_mpz_t): the multivariate gcd and factorization. It also implements rational fractions of differential polynomials. The baz library relies on the ba0, bav and bap libraries.

7.1.1 Global variables and settings

Setting variables are global configuration variables which permit to tune the behaviour of the library functions. This tuning is supposed to be performed before baz_restart is called (more generally, outside any sequence of calls to the library). They are bundled in the baz_initialized_global global variable which has the following type.

Description of type baz_initialized_global. This type declaration belongs to baz_global.h.

```
struct baz_initialized_global
{
  struct
  {
/*
 * If nonzero, the lhs of prolongation patterns are quoted using
 * its first character
 */
    char *lhs_quotes;
  } prolongation_pattern;
};
```

void baz_reset_all_settings (void) [Function]

Reset all settings variables to their default values. This function must be called outside sequences of calls to the library. This function belongs to baz_common.c. It is run by 65 tests, including 0 in the baz library and 65 in the DifferentialAlgebra package.

7.1.2 Basic types, constants and naming conventions

The name of every exported object defined in the library starts by baz_. Functions starting by baz_init initialize raw data structures (constructors). Functions starting by baz_new dynamically allocate data structures and apply the corresponding baz_init functions on them. Functions starting by baz_reset reset data structures which are assumed to be already initialized. Functions starting by baz_set perform assignments and systematically copy the sources to the destinations. A *sequence of calls* to the baz library starts with a call to baz_restart and finishes by a call to baz_terminate.

void baz_restart (ba0_int_p *time_limit*, ba0_int_p *memory_limit*) [Function]

Call the bap_restart function with the same parameters and defines a few more formats. This function belongs to baz_common.c. It is run by 359 tests, including 79 in the baz library and 65 in the DifferentialAlgebra package.

`void baz_terminate (enum ba0_restart_level level)` [Function]
Call the `bap_terminate` function with the same parameter. This function belongs to `baz_common.c`. It is run by 358 tests, including 78 in the `baz` library and 65 in the `DifferentialAlgebra` package.

7.2 Primitives

The key algorithms are the gcd and the factorization of polynomials over the integers. They are much inspired from the computer algebra book of Geddes, Czapor and Labahn.

7.2.1 The gcd and related methods

`void baz_gcd_pseudo_division_polynom_mpz (struct bap_polynom_mpz *Q, struct bap_polynom_mpz *R, struct bap_product_mpz *H, struct bap_polynom_mpz *A, struct bap_polynom_mpz *B, struct bav_variable *x)` [Function]
Variant of the pseudo division algorithm where, at each step, the current remainder is multiplied by the smallest needed factor. This factor is determined by a gcd computation. The product H receives the product by which A was multiplied during the process. Parameters Q, R and H may be the zero pointer. The variable x may be zero. In that case, it is understood to be the leading variable of B. This function belongs to `baz_polyspec_mpz.c`. It is run by 204 tests, including 2 in the `baz` library and 33 in the `DifferentialAlgebra` package.

`void baz_gcd_prem_polynom_mpz (struct bap_polynom_mpz *R, struct bap_product_mpz *H, struct bap_polynom_mpz *A, struct bap_polynom_mpz *B, struct bav_variable *x)` [Function]
Variant of `baz_gcd_pseudo_division_polynom_mpz`. This function belongs to `baz_polyspec_mpz.c`. It is run by 200 tests, including 0 in the `baz` library and 33 in the `DifferentialAlgebra` package.

`void baz_gcd_pquo_polynom_mpz (struct bap_polynom_mpz *Q, struct bap_product_mpz *H, struct bap_polynom_mpz *A, struct bap_polynom_mpz *B, struct bav_variable *x)` [Function]
Variant of `baz_gcd_pseudo_division_polynom_mpz`. This function belongs to `baz_polyspec_mpz.c`. It is not run by any test.

`void baz_genpoly_polynom_mpz (struct bap_polynom_mpz *R, struct bap_polynom_mpz *A, ba0_mpz_t n, struct bav_variable *v)` [Function]
Assume $A \in Z[x_1, \ldots, x_\ell]$. Assigns to R the unique polynomial of $Z[x_1, \ldots, x_\ell, v]$ such that R is equal to A modulo $(v - n)$ and maxnorm of R is less than $n/2$. Exception `BA0_ERRALG` is raised if n is even. This function belongs to `baz_polyspec_mpz.c`. It is run by 57 tests, including 11 in the `baz` library and 4 in the `DifferentialAlgebra` package.

`void baz_gcd_univariate_tableof_polynom_mpz (struct bap_polynom_mpz *G, struct bap_tableof_polynom_mpz *polys)` [Function]
Assign to G the gcd of all the elements of *polys*. All polynomials are assumed to be univariate, depending on the same variable. Exception `BA0_ERRALG` is raised if *polys*

is empty. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 3 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void baz_gcdheu_tableof_polynom_mpz (struct bap_polynom_mpz *G, struct bap_tableof_polynom_mpz *polys0, ba0_int_p toocomplex)` [Function]

Assign to G the gcd of the polynomials in *polys0*, computed by means of heuristic gcd algorithm. The function may fail (possibly because of a too complex computation). In that case, it raises the exception `BAP_ERRHEU`. The parameter *toocomplex* should provide an estimation of what a too complex computation is (current values are picked in the range [4000,10000]). This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 58 tests, including 12 in the `baz` library and 4 in the `DifferentialAlgebra` package.

`void baz_extended_Zassenhaus_gcd_tableof_polynom_mpz (struct bap_product_mpz *G, struct bap_tableof_polynom_mpz *polys0, bool giveup)` [Function]

Assign to G the gcd of the polynomials in *polys0*, computed using the extended Zassenhaus gcd algorithm. If *giveup* is `true` then the gcd algorithms (recursively called by this function) which are considered as the most costly are not called and the function may fail to compute a complete gcd. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 3 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void baz_gcd_tableof_polynom_mpz (struct bap_product_mpz *G, struct bap_tableof_polynom_mpz *polys, bool giveup)` [Function]

Assign to G the gcd of the polynomials in *polys0*. This function relies on many different algorithms. If *giveup* is `true` then the algorithms which are considered as the most costly are not called and the function may fail to compute a complete gcd. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 329 tests, including 70 in the `baz` library and 53 in the `DifferentialAlgebra` package.

`void baz_gcd_polynom_mpz (struct bap_polynom_mpz *G, struct bap_polynom_mpz *cofA, struct bap_polynom_mpz *cofB, struct bap_polynom_mpz *A, struct bap_polynom_mpz *B)` [Function]

Assign to G the gcd of the polynomials A and B. The polynomials *cofA* and *cofB* are assigned A/G and B/G. The parameters G, *cofA* and *cofB* may be the zero pointer. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 306 tests, including 49 in the `baz` library and 53 in the `DifferentialAlgebra` package.

`void baz_content_polynom_mpz (struct bap_polynom_mpz *R, struct bap_polynom_mpz *A, struct bav_variable *v)` [Function]

Assign to R the content of A with respect to variable v or its leader, if v is zero. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 187 tests, including 5 in the `baz` library and 34 in the `DifferentialAlgebra` package.

`void baz_primpart_polynom_mpz (struct bap_polynom_mpz *R, struct bap_polynom_mpz *A, struct bav_variable *v)` [Function]

Assign to R the primitive part of A with respect to variable v or its leader, if v is zero. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 180 tests, including 0 in the `baz` library and 32 in the `DifferentialAlgebra` package.

`void baz_content_tableof_polynom_mpz (struct bap_product_mpz *G, struct bap_tableof_polynom_mpz *polys0, struct bav_variable *x, bool giveup)` [Function]

Assign to G the gcd of the coefficients of the elements of *polys0* with respect to v (or the highest derivative occurring in *polys0*, if v is zero). This function relies on `baz_gcd_tableof_polynom_mpz`. If *giveup* is `true` then the algorithms which are considered as the most costly are not called and the function may fail to compute a complete gcd. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 226 tests, including 23 in the `baz` library and 36 in the `DifferentialAlgebra` package.

7.2.2 The factorization and related methods

`void baz_Yun_polynom_mpz (struct bap_product_mpz *prod, struct bap_polynom_mpz *P0, bool giveup)` [Function]

Assign to *prod* the squarefree decomposition of *P0* in the sense of Yun, with respect to its leading derivative. If *giveup* is `true` then the algorithms which are considered as the most costly are not called and the function may fail to compute a complete gcd. This function relies on `baz_gcd_tableof_polynom_mpz`. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 198 tests, including 18 in the `baz` library and 33 in the `DifferentialAlgebra` package.

`void baz_squarefree_polynom_mpz (struct bap_product_mpz *prod, struct bap_polynom_mpz *A)` [Function]

Call `baz_Yun_polynom_mpz` with *giveup* equal to `false`. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 49 tests, including 16 in the `baz` library and 9 in the `DifferentialAlgebra` package.

`void baz_factor_easy_polynom_mpz (struct bap_product_mpz *prod, struct ba0_tableof_bool *keep, struct bap_polynom_mpz *A0, struct bap_listof_polynom_mpz *F0)` [Function]

Assign to *prod* a factorization of *A0* which is not too difficult to compute and not necessarily irreducible. Eventually, if a factor of *prod* has a nontrivial gcd with some element of *F0* then it divides it. The argument *keep* is assigned a table of the same size as *prod*. The ith element of *keep* is `true` if and only if the ith factor of *prod* does not divides any element of *F0*. The `false` entries appear in the leftmost part of *keep*, the `true` entries in the rightmost part. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 214 tests, including 4 in the `baz` library and 33 in the `DifferentialAlgebra` package.

`void baz_factor_easy_product_mpz (struct bap_product_mpz *prod, struct ba0_tableof_bool *keep, struct bap_product_mpz *prod0, struct bap_listof_polynom_mpz *F0)` [Function]

Assign to *prod* a factorization of *prod0* which is not too difficult to compute and not necessarily irreducible. Eventually, if a factor of *prod* has a nontrivial gcd with some element of *F0* then it divides it. The argument *keep* is assigned a table of the same size as *prod*. The ith element of *keep* is `true` if and only if the ith factor of *prod* does not divides any element of *F0*. The `false` entries appear in the leftmost part of *keep*, the `true` entries in the rightmost part. This function belongs to `baz_gcd_polynom_mpz.c`. It is run by 147 tests, including 2 in the `baz` library and 30 in the `DifferentialAlgebra` package.

`void baz_factor_polynom_mpz (struct bap_product_mpz *prod, struct bap_polynom_mpz *A)` [Function]

Assign to *prod* the complete factorization of A over the integers. The numerical factor of *prod* receives the gcd of the numerical coefficients of A and carries the sign of the leading coefficient. This function belongs to `baz_factor_polynom_mpz.c`. It is run by 4 tests, including 4 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void baz_factor_polynom_mpq (struct bap_product_mpq *R, struct bap_polynom_mpq *A)` [Function]

Assign to R the complete factorization of A over the ring of integer numbers. This function belongs to `baz_factor_polynom_mpq.c`. It is run by 1 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

7.2.3 Miscellaneous primitives

`void baz_monomial_reduce_polynom_mpz (struct bap_polynom_mpz *Q, struct bap_polynom_mpz *R, ba0_mpz_t c, struct bap_polynom_mpz *A, struct bap_polynom_mpz *B)` [Function]

Given A and B (B being non-numeric), compute a relation $c\,A = B\,Q + R$ with R reduced (in the sense of the Gröbner basis theory) w.r.t. B. This function belongs to `baz_polyspec_mpz.c`. It is run by 1 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void baz_yet_another_point_int_p_mpz (struct ba0_point_int_p *point, struct bap_tableof_polynom_mpz *nonzero, struct bap_product_mpz *prod, struct bav_variable *avoid)` [Function]

Assign to *point* a new evaluation point which does not annihilate any of the polynomials in nonzero and does not evaluate any factor of prod to $0, \pm 1, \pm 2$. All variables present in *nonzero* and *prod* must occur in *point*. At least one of the variables present in *nonzero* or *prod*, and one of the variables which are not present in *nonzero* or *prod*, are guaranteed to be modified (if any). The variable avoid is not touched. It is allowed to be `BAV_NOT_A_VARIABLE`. Exception `BA0_ERRALG` is raised if no value can be modified. This function belongs to `baz_polyspec_mpz.c`. It is run by 101 tests, including 22 in the `baz` library and 19 in the `DifferentialAlgebra` package.

`void baz_eval_to_ratfrac_at_point_ratfrac_polynom_mpz` [Function]
`(struct baz_ratfrac *R, struct bap_polynom_mpz *A, struct baz_point_ratfrac *point)`

This function belongs to `baz_eval_polyspec_mpz.c`. It is run by 113 tests, including 15 in the `baz` library and 11 in the `DifferentialAlgebra` package.

7.2.4 Primitives applying to products

`void baz_factor_numeric_content_product_mpz (struct bap_product_mpz *R, struct bap_product_mpz *A)` [Function]

Assign to R the product obtained by replacing each factor of A by its sign-normalized numerical primitive. Some factors of A may get collected in R. This function belongs to `baz_prospec_mpz.c`. It is run by 95 tests, including 0 in the `baz` library and 24 in the `DifferentialAlgebra` package.

`void baz_gcd_product_mpz (struct bap_product_mpz *G, struct bap_product_mpz *cofA, struct bap_product_mpz *cofB, struct bap_product_mpz *A, struct bap_product_mpz *B)` [Function]

Assign to G the gcd of the products A and B. The products *cofA* and *cofB* are assigned A/G and B/G. The parameters G, *cofA* and *cofB* may be the zero pointer. The result variables have their numerical contents factored. This function belongs to `baz_prospec_mpz.c`. It is run by 95 tests, including 0 in the `baz` library and 24 in the `DifferentialAlgebra` package.

7.2.5 Isolation of real roots

Description of type `baz_typeof_realroot_interval`. This data type permits to control the behaviour of algorithms for isolating real roots of univariate polynomials. This type declaration belongs to `baz_realroot_mpq.h`.

```
enum baz_typeof_realroot_interval
{
// Output intervals whenever they have width < epsilon
  baz_any_interval,
// Output intervals if they are guaranteed to isolate exactly one root
  baz_isolation_interval
};
```

`void baz_positive_roots_polynom_mpq (struct ba0_tableof_interval_mpq *T, struct bap_polynom_mpq *P0, enum baz_typeof_realroot_interval type, ba0_mpq_t epsilon)` [Function]

Consider first the case where *type* is `baz_isolation_interval`. Append to T intervals which are either open or closed. Open intervals are disjoint and have width less than *epsilon*. Each closed interval contains a single rational number. Each interval isolates exactly one positive root of *P0*. Each positive root of *P0* is isolated in exactly one interval.

Consider now the case where *type* is `baz_any_interval`. Open intervals are disjoint and have width less than *epsilon*. Each positive root of *P0* is contained in some interval. However, an open interval may not contain any root of *P0* or contain more than one root.

The polynomial *P0* must be a univariate polynomial.

The resulting table T is sorted by increasing order. This function belongs to `baz_realroot_mpq.c`. It is run by 45 tests, including 12 in the `baz` library and 9 in the `DifferentialAlgebra` package.

`void baz_positive_integer_roots_polynom_mpq (struct ba0_tableof_mpz *T, struct bap_polynom_mpq *P)` [Function]

Assign to T the positive integer roots of P, sorted by increasing value. This function belongs to `baz_realroot_mpq.c`. It is run by 40 tests, including 7 in the `baz` library and 9 in the `DifferentialAlgebra` package.

`void baz_positive_integer_roots_polynom_mpz (struct ba0_tableof_mpz *T, struct bap_polynom_mpz *P, struct bav_variable *v)` [Function]

Assign to T the positive integer roots of P, viewed as a univariate polynomial in v. The polynomial P may depend on other variables than v. If v is `BAV_NOT_A_VARIABLE`, it is supposed to be the leader of P. The roots are sorted by increasing value. This function belongs to `baz_realroot_mpq.c`. It is run by 4 tests, including 4 in the `baz` library and 0 in the `DifferentialAlgebra` package.

7.3 Rational fractions with `mpz_t` coefficients

Description of type `baz_ratfrac`. This data type implements fractions of differential polynomials. The fraction is not necessarily reduced. The denominator is nonzero. This type declaration belongs to `baz_ratfrac.h`.

```
struct baz_ratfrac
{
  struct bap_polynom_mpz numer;
  struct bap_polynom_mpz denom;
};
```

`void baz_init_ratfrac (struct baz_ratfrac *R)` [Function]

Initialize R to zero. This function belongs to `baz_ratfrac.c`. It is run by 308 tests, including 38 in the `baz` library and 62 in the `DifferentialAlgebra` package.

`void baz_init_readonly_ratfrac (struct baz_ratfrac *R)` [Function]

Initialize R to zero in readonly mode (note: the denominator is not readonly). This function belongs to `baz_ratfrac.c`. It is run by 114 tests, including 15 in the `baz` library and 12 in the `DifferentialAlgebra` package.

`struct baz_ratfrac * baz_new_ratfrac (void)` [Function]

Allocate a new ratfrac, initialize it and return it. This function belongs to `baz_ratfrac.c`. It is run by 224 tests, including 22 in the `baz` library and 48 in the `DifferentialAlgebra` package.

`struct baz_ratfrac * baz_new_readonly_ratfrac (void)` [Function]

Allocate a new ratfrac, initialize it in readonly mode and return it. This function belongs to `baz_ratfrac.c`. It is run by 111 tests, including 13 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_set_ratfrac (struct baz_ratfrac *A, struct baz_ratfrac *B)` [Function]

Assign B to A. This function belongs to `baz_ratfrac.c`. It is run by 254 tests, including 38 in the `baz` library and 62 in the `DifferentialAlgebra` package.

`void baz_set_tableof_ratfrac (struct baz_tableof_ratfrac *dst, struct baz_tableof_ratfrac *src)` [Function]

Assign *src* to *dst*. Entries of *src* are allowed to be `BAZ_NOT_A_RATFRAC`. This function belongs to `baz_ratfrac.c`. It is run by 51 tests, including 0 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`void baz_set_tableof_tableof_ratfrac (struct baz_tableof_tableof_ratfrac *dst, struct baz_tableof_tableof_ratfrac *src)` [Function]

Assign *src* to *dst*. This function belongs to `baz_ratfrac.c`. It is run by 38 tests, including 0 in the `baz` library and 8 in the `DifferentialAlgebra` package.

`void baz_set_tableof_tableof_tableof_ratfrac (struct baz_tableof_tableof_tableof_ratfrac *dst, struct baz_tableof_tableof_tableof_ratfrac *src)` [Function]

Assign *src* to *dst*. This function belongs to `baz_ratfrac.c`. It is not run by any test.

`void baz_set_ratfrac_zero (struct baz_ratfrac *A)` [Function]

Assign 0 to A. This function belongs to `baz_ratfrac.c`. It is run by 88 tests, including 16 in the `baz` library and 18 in the `DifferentialAlgebra` package.

`void baz_set_ratfrac_one (struct baz_ratfrac *A)` [Function]

Assign 1 to A. This function belongs to `baz_ratfrac.c`. It is run by 112 tests, including 14 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_set_ratfrac_term (struct baz_ratfrac *R, struct bav_term *T)` [Function]

Assign T to R. This function belongs to `baz_ratfrac.c`. It is run by 1 tests, including 0 in the `baz` library and 1 in the `DifferentialAlgebra` package.

`void baz_set_ratfrac_polynom_mpz (struct baz_ratfrac *R, struct bap_polynom_mpz *A)` [Function]

Assign A to R. This function belongs to `baz_ratfrac.c`. It is run by 254 tests, including 38 in the `baz` library and 62 in the `DifferentialAlgebra` package.

`void baz_add_ratfrac_polynom_mpz (struct baz_ratfrac *R, struct baz_ratfrac *A, struct bap_polynom_mpz *P)` [Function]

Assign $A + P$ to R. This function belongs to `baz_ratfrac.c`. It is not run by any test.

`void baz_mul_ratfrac_polynom_mpz (struct baz_ratfrac *R, struct baz_ratfrac *A, struct bap_polynom_mpz *P)` [Function]

Assign $A\,P$ to R. This function belongs to `baz_ratfrac.c`. It is run by 111 tests, including 13 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_mul_ratfrac_polynom_mpq (struct baz_ratfrac *R, struct baz_ratfrac *A, struct bap_polynom_mpq *P)` [Function]

Assign $A\,P$ to R. This function belongs to `baz_ratfrac.c`. It is run by 53 tests, including 0 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`void baz_set_ratfrac_fraction (struct baz_ratfrac *R, struct bap_polynom_mpz *P, struct bap_polynom_mpz *Q)` [Function]

Assign P/Q to R. This function belongs to `baz_ratfrac.c`. It is run by 8 tests, including 4 in the `baz` library and 3 in the `DifferentialAlgebra` package.

`bool baz_is_zero_ratfrac (struct baz_ratfrac *R)` [Function]

Return `true` if R if 0. This function belongs to `baz_ratfrac.c`. It is run by 252 tests, including 37 in the `baz` library and 61 in the `DifferentialAlgebra` package.

`bool baz_is_one_ratfrac (struct baz_ratfrac *R)` [Function]

Return `true` if R is 1. This function belongs to `baz_ratfrac.c`. It is not run by any test.

`bool baz_is_numeric_ratfrac (struct baz_ratfrac *R)` [Function]

Return `true` if the numerator and the denomnator of R are numerical. This function belongs to `baz_ratfrac.c`. It is run by 9 tests, including 4 in the `baz` library and 4 in the `DifferentialAlgebra` package.

`bool baz_is_constant_ratfrac (struct baz_ratfrac *A, struct bav_symbol *s)` [Function]

Return `true` if the derivative of A with respect to s is zero. This function belongs to `baz_ratfrac.c`. It is run by 1 tests, including 0 in the `baz` library and 1 in the `DifferentialAlgebra` package.

`bool baz_equal_ratfrac (struct baz_ratfrac *A, struct baz_ratfrac *B)` [Function]

Return `true` if A and B have the same numerator and the same denominator. This function belongs to `baz_ratfrac.c`. It is run by 1 tests, including 0 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`bool baz_depend_ratfrac (struct baz_ratfrac *A, struct bav_variable *v)` [Function]

Return `true` if A depends on v. This function belongs to `baz_ratfrac.c`. It is run by 9 tests, including 4 in the `baz` library and 4 in the `DifferentialAlgebra` package.

`void baz_numer_ratfrac (struct bap_polynom_mpz *A, struct baz_ratfrac *R)` [Function]

Assign to A the numerator of R. This function belongs to `baz_ratfrac.c`. It is run by 149 tests, including 0 in the `baz` library and 7 in the `DifferentialAlgebra` package.

`void baz_denom_ratfrac (struct bap_polynom_mpz *A, struct baz_ratfrac *R)` [Function]

Assign to A the denominator of R. This function belongs to `baz_ratfrac.c`. It is not run by any test.

`struct bav_variable * baz_leader_ratfrac (struct baz_ratfrac *R)` [Function]

Return the highest variable occurring in the numerator and the denominator of A. The rational fraction is supposed to be reduced. Exception `BAP_ERRCST` is raised if R is numerical. This function belongs to `baz_ratfrac.c`. It is run by 18 tests, including 7 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`struct bav_rank baz_rank_ratfrac (struct baz_ratfrac *R)` [Function]

Return the rank of A, which is defined as follows: if A is numeric, then its rank is the constant rank (or the rank of zero), else it is v^{n-d} where v, n and d denote the leader of A, the degree of its numerator in v and the degree of its denominator in v. When $n = d$, the rank of A is the constant rank. This function belongs to `baz_ratfrac.c`. It is run by 9 tests, including 4 in the `baz` library and 5 in the `DifferentialAlgebra` package.

`int baz_compare_rank_ratfrac (const void *PP, const void *QQ)` [Function]

A comparison function for sorting tables of rational fractions by increasing rank, using `qsort`. This function belongs to `baz_ratfrac.c`. It is not run by any test.

`void baz_mark_indets_ratfrac (struct bav_dictionary_variable *dict, struct bav_tableof_variable *vars, struct baz_ratfrac *R)` [Function]

Append to *vars* the variables occurring in R which are not already present in *vars*. Each element of *vars* is supposed to be registered in the dictionary *dict*. This function belongs to `baz_ratfrac.c`. It is run by 99 tests, including 2 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`void baz_initial_ratfrac (struct baz_ratfrac *init, struct baz_ratfrac *R)` [Function]

Assign to *init* the initial of R in readonly mode. Raises the exception `BAP_ERRNHL` if the rank of R has negative degree. This function belongs to `baz_ratfrac.c`. It is run by 4 tests, including 0 in the `baz` library and 4 in the `DifferentialAlgebra` package.

`void baz_reductum_ratfrac (struct baz_ratfrac *reductum, struct baz_ratfrac *R)` [Function]

Assign to *reductum* the reductum of R in readonly mode. Raises the exception `BAP_ERRNHL` if the rank of R has negative degree. This function belongs to `baz_ratfrac.c`. It is run by 1 tests, including 0 in the `baz` library and 1 in the `DifferentialAlgebra` package.

`void baz_lcoeff_and_reductum_ratfrac (struct baz_ratfrac *lcoeff, struct baz_ratfrac *reductum, struct baz_ratfrac *R, struct bav_variable *v)` [Function]

Assign to *lcoeff* and *reductum* the leading coefficient and the reductum of R with respect to v. The variable v does not need to be greater than or equal to the leading variable of R. If v is the zero pointer, it is taken to be the leading variable of R. The rational fraction R may be a numerical rational fraction. In that case, R is assigned

to *lcoeff* and zero is assigned to *reductum*. The resulting rational fractions are plain rational fractions (not readonly). Exception `BAP_ERRNHL` is raised if v is zero and the rank of R has negative degree. Exception `BAP_ERRVPD` is raised if v is nonzero and is present in the denominator of R. This function belongs to `baz_ratfrac.c`. It is run by 2 tests, including 0 in the `baz` library and 2 in the `DifferentialAlgebra` package.

`void baz_sort_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A)` [Function]

Sort A with respect to the current ordering. The result, which is readonly, is stored in R. This function belongs to `baz_ratfrac.c`. It is run by 114 tests, including 15 in the `baz` library and 12 in the `DifferentialAlgebra` package.

`void baz_physort_ratfrac (struct baz_ratfrac *R)` [Function]

Apply `bap_physort_polynom_mpz` over the numerator and the denominator of R. This function belongs to `baz_ratfrac.c`. It is run by 114 tests, including 15 in the `baz` library and 12 in the `DifferentialAlgebra` package.

`void baz_normalize_numeric_initial_ratfrac (struct baz_ratfrac *R)` [Function]

Make the numeric leading coefficient of the denominator of R positive. This function belongs to `baz_ratfrac.c`. It is run by 232 tests, including 37 in the `baz` library and 53 in the `DifferentialAlgebra` package.

`void baz_reduce_numeric_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A)` [Function]

Divide the numerator and the denominator of A by the gcd of their numerical content. Result in R. The numerical leading coefficient of the denominator of R is positive. This function belongs to `baz_ratfrac.c`. It is run by 247 tests, including 34 in the `baz` library and 59 in the `DifferentialAlgebra` package.

`void baz_reduce_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A)` [Function]

Assign to R the reduced fraction A. The gcd of the numerator and the denominator of A is factored out. The numerical leading coefficient of the denominator of R is positive. This function belongs to `baz_ratfrac.c`. It is run by 253 tests, including 38 in the `baz` library and 61 in the `DifferentialAlgebra` package.

`void baz_neg_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A)` [Function]

Assign $-A$ to R. This function belongs to `baz_ratfrac.c`. It is not run by any test.

`void baz_add_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A, struct baz_ratfrac *B)` [Function]

Assign $A + B$ to R. This function belongs to `baz_ratfrac.c`. It is run by 208 tests, including 34 in the `baz` library and 51 in the `DifferentialAlgebra` package.

`void baz_sub_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A, struct baz_ratfrac *B)` [Function]

Assign $A - B$ to R. This function belongs to `baz_ratfrac.c`. It is run by 211 tests, including 27 in the `baz` library and 48 in the `DifferentialAlgebra` package.

`void baz_mul_ratfrac_numeric (struct baz_ratfrac *R, struct baz_ratfrac *A, ba0_mpz_t c)` [Function]

Assign $c\,A$ to R. This function belongs to `baz_ratfrac.c`. It is run by 57 tests, including 1 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_mul_ratfrac_numeric_mpq (struct baz_ratfrac *R, struct baz_ratfrac *A, ba0_mpq_t q)` [Function]

Assign $q\,A$ to R. This function belongs to `baz_ratfrac.c`. It is run by 57 tests, including 1 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_mul_ratfrac_variable (struct baz_ratfrac *R, struct baz_ratfrac *A, struct bav_variable *v, bav_Idegree d)` [Function]

Assign $A\,v^d$ to R. This function belongs to `baz_ratfrac.c`. It is run by 3 tests, including 3 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void baz_mul_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A, struct baz_ratfrac *B)` [Function]

Assign $A\,B$ to R. This function belongs to `baz_ratfrac.c`. It is run by 232 tests, including 37 in the `baz` library and 53 in the `DifferentialAlgebra` package.

`void baz_invert_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A)` [Function]

Assign $1/A$ to R. This function belongs to `baz_ratfrac.c`. It is run by 3 tests, including 0 in the `baz` library and 3 in the `DifferentialAlgebra` package.

`void baz_div_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A, struct baz_ratfrac *B)` [Function]

Assign A/B to R. This function belongs to `baz_ratfrac.c`. It is run by 86 tests, including 35 in the `baz` library and 23 in the `DifferentialAlgebra` package.

`void baz_pow_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A, bav_Idegree d)` [Function]

Assign A^d to R. The exponent d is allowed to be negative. This function belongs to `baz_ratfrac.c`. It is run by 186 tests, including 26 in the `baz` library and 49 in the `DifferentialAlgebra` package.

`void baz_separant_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A)` [Function]

Assign the separant of A (i.e. its derivative with respect to its leader) to R. This function belongs to `baz_ratfrac.c`. It is run by 11 tests, including 6 in the `baz` library and 4 in the `DifferentialAlgebra` package.

`void baz_separant2_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A, struct bav_variable *v)` [Function]

Assign $\partial A/\partial v$ to R. This function belongs to `baz_ratfrac.c`. It is run by 9 tests, including 4 in the `baz` library and 4 in the `DifferentialAlgebra` package.

`void baz_diff_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A, struct bav_symbol *s)` [Function]

Assign to R the rational fraction obtained by differentiating A with respect to s. This function belongs to `baz_ratfrac.c`. It is run by 19 tests, including 6 in the `baz` library and 12 in the `DifferentialAlgebra` package.

`void baz_rat_bilge_mpz (struct baz_ratfrac *result, struct baz_ratfrac *whatsleft, struct baz_ratfrac *F0, struct bav_symbol *x)` [Function]

Compute two rational differential fractions R and W such that the derivative of R w.r.t. x plus W is equal to F. If F is the derivative of some rational differential fraction, then W is zero. This algorithm is mostly due to François Lemaire. This function belongs to `baz_rat_bilge_mpz.c`. It is run by 8 tests, including 4 in the `baz` library and 3 in the `DifferentialAlgebra` package.

`void baz_collect_terms_tableof_ratfrac (struct baz_tableof_ratfrac *collected_coeffs, struct baz_tableof_ratfrac *collected_terms, struct baz_tableof_ratfrac *coeffs, struct baz_tableof_ratfrac *terms)` [Function]

The table *coeffs* and *terms* are two tables of the same size. They represent the sum of the products of the coefficients by the terms, pairwise. The rational fractions are supposed to be reduced. The function collects together the terms, whose coefficients are equal, up to some multiplicative rational number factor. Mathematically, the represented sum is not changed. This function belongs to `baz_collect_terms_ratfrac.c`. It is run by 1 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void baz_eval_to_polynom_at_point_int_p_ratfrac (struct bap_polynom_mpq *R, struct baz_ratfrac *A, struct ba0_point_int_p *point)` [Function]

Assign A mod *point* to R. Exception `BA0_ERRALG` is raised if some variables of the denominator of A do not get evaluated. Exception `BA0_ERRIVZ` is raised if the denominator of A evaluates to zero. This function belongs to `baz_eval_ratfrac.c`. It is run by 36 tests, including 1 in the `baz` library and 9 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p baz_sizeof_ratfrac (struct baz_ratfrac *F, enum ba0_garbage_code code)` [Function]

Return the size of the memory needed to perform a copy of F. If *code* is `ba0_embedded` then A is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `baz_ratfrac.c`. It is run by 11 tests, including 0 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p baz_sizeof_tableof_ratfrac (struct baz_tableof_ratfrac *T, enum ba0_garbage_code code)` [Function]

Return the size of the memory needed to perform a copy of T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not

taken into account. This function belongs to `baz_ratfrac.c`. It is run by 11 tests, including 0 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p baz_sizeof_tableof_tableof_ratfrac (struct baz_tableof_tableof_ratfrac *T, enum ba0_garbage_code code)` [Function]

Return the size of the memory needed to perform a copy of T. If *code* is `ba0_embedded` then T is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `baz_ratfrac.c`. It is not run by any test.

`void baz_switch_ring_ratfrac (struct baz_ratfrac *A, struct bav_differential_ring *R)` [Function]

This low level function should be used in conjunction with `bav_set_differential_ring`: if R is a ring obtained by application of `bav_set_differential_ring` to the ring A refers to, then this function makes A refer to R. This function belongs to `baz_ratfrac.c`. It is run by 11 tests, including 0 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`void * baz_scanf_ratfrac (void *R)` [Function]

The general parsing function of rational fractions. It is called by `ba0_scanf/%Qz`. This function does not simplify to zero the zero derivatives of parameters. The fact that derivations commute is not taken into account. See `baz_scanf_simplify_ratfrac`. This function belongs to `baz_ratfrac.c`. It is run by 206 tests, including 38 in the `baz` library and 48 in the `DifferentialAlgebra` package.

`void * baz_scanf_simplify_ratfrac (void *R)` [Function]

Parsing functions for rational fractions. Zero derivatives of parameters are simplified to zero. The fact that derivations commute is taken into account. This function is called by `ba0_scanf/%simplify_Qz`. This function belongs to `baz_ratfrac.c`. It is run by 161 tests, including 0 in the `baz` library and 35 in the `DifferentialAlgebra` package.

`void * baz_scanf_expanded_ratfrac (void *AA)` [Function]

Parsing functions for rational fractions. The numerator and the denominator are supposed to be expanded polynomials over the rational numbers. This function does not simplify to zero the zero derivatives of parameters. The fact that derivations commute is not taken into account. See `baz_scanf_simplify_expanded_ratfrac`. This function is called by `ba0_scanf/%expanded_Qz`. This function belongs to `baz_ratfrac.c`. It is run by 1 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void * baz_scanf_simplify_expanded_ratfrac (void *R)` [Function]

Parsing functions for rational fractions. The numerator and the denominator are supposed to be expanded polynomials over the rational numbers. This function simplifies to zero the zero derivatives of parameters. The fact that derivations commute is taken into account. This function is called by `ba0_scanf/%simplify_expanded_Qz`. This function belongs to `baz_ratfrac.c`. It is run by 4 tests, including 2 in the `baz` library and 2 in the `DifferentialAlgebra` package.

`void baz_printf_ratfrac (void *AA)` [Function]
The general printing function for rational fractions. It is called by `ba0_printf/%Qz`. This function belongs to `baz_ratfrac.c`. It is run by 115 tests, including 28 in the `baz` library and 27 in the `DifferentialAlgebra` package.

7.3.1 Relational expressions between rational fractions

Description of type `baz_typeof_relop`. This data type provides an encoding for relational operators. This type declaration belongs to `baz_rel_ratfrac.h`.

```
enum baz_typeof_relop
{
  baz_none_relop,
  baz_equal_relop,
  baz_not_equal_relop,
  baz_greater_relop,
  baz_greater_or_equal_relop,
  baz_less_relop,
  baz_less_or_equal_relop
};
```

Description of type `baz_rel_ratfrac`. This data type implements a pair of rational fractions connected by a relational operator. This type declaration belongs to `baz_rel_ratfrac.h`.

```
struct baz_rel_ratfrac
{
  struct baz_ratfrac lhs;
  struct baz_ratfrac rhs;
  enum baz_typeof_relop op;
};
```

`void baz_init_rel_ratfrac (struct baz_rel_ratfrac *R)` [Function]
Initialize R to zero. This function belongs to `baz_rel_ratfrac.c`. It is run by 41 tests, including 1 in the `baz` library and 36 in the `DifferentialAlgebra` package.

`struct baz_rel_ratfrac * baz_new_rel_ratfrac (void)` [Function]
Allocate a new `struct baz_rel_ratfrac *`, initialize it and return it. This function belongs to `baz_rel_ratfrac.c`. It is run by 37 tests, including 1 in the `baz` library and 36 in the `DifferentialAlgebra` package.

`void baz_set_rel_ratfrac (struct baz_rel_ratfrac *dst, struct baz_rel_ratfrac *src)` [Function]
Assign *src* to *dst*. This function belongs to `baz_rel_ratfrac.c`. It is not run by any test.

`void baz_set_ratfrac_rel_ratfrac (struct baz_ratfrac *Q, struct baz_rel_ratfrac *R)` [Function]
Subtract the right handside from the left handside of R and assign the result to Q. This function belongs to `baz_rel_ratfrac.c`. It is run by 36 tests, including 0 in the `baz` library and 36 in the `DifferentialAlgebra` package.

`void * baz_scanf_rel_ratfrac (void *A)` [Function]
Parsing function for `struct baz_rel_ratfrac *`. This function is called by `ba0_scanf/%relQz`. This function belongs to `baz_rel_ratfrac.c`. It is run by 41 tests, including 1 in the `baz` library and 36 in the `DifferentialAlgebra` package.

`void baz_printf_rel_ratfrac (void *A)` [Function]

Printing function for `struct baz_rel_ratfrac *`. This function is called by `ba0_printf/%relQz`. This function belongs to `baz_rel_ratfrac.c`. It is run by 5 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

7.4 Prolongation patterns

Description of type `baz_prolongation_pattern`. This data type implements patterns for prolongating points. Here is an example of pattern:

```
R       = DifferentialRing (derivations = [x], blocks = [y,z])
pattern = { y[(x,k)] : 'y[k]/factorial(k)' }
```

Every derivative of y matches the pattern. The pattern associates $y_k/k!$ to the kth derivative of y for every nonnegative integer k.

The field `dict` only aims at speeding up the evaluation of replacement values. It is provided to the substitution dictionary of the lexical analyzer. This type declaration belongs to `baz_prolongation_pattern.h`.

```
struct baz_prolongation_pattern
{
// The dependent variables (such as "y") matching the pattern
  struct bav_tableof_symbol deps;
// The identifiers denoting the orders of derivation (such as "k")
// The table follows the order provided by bav_global.R.ders
  struct ba0_tableof_string idents;
// The replacement values (such as "y[k]/factorial(k)")
  struct ba0_tableof_string exprs;
// A dictionary which maps identifiers (such as "k") to the corresponding
//      entry in idents (used by the substitution dictionary)
  struct ba0_dictionary_string dict;
};
```

`void baz_set_settings_prolongation_pattern (char *lhs_quotes)` [Function]

Assign *lhs_quotes* to the corresponding setting variable. This function belongs to `baz_prolongation_pattern.c`. It is run by 65 tests, including 0 in the `baz` library and 65 in the `DifferentialAlgebra` package.

`void baz_get_settings_prolongation_pattern (char **lhs_quotes)` [Function]

Assign to **lhs_quotes* the value of the corresponding setting variable. The argument may be zero. This function belongs to `baz_prolongation_pattern.c`. It is not run by any test.

`void baz_init_prolongation_pattern (struct baz_prolongation_pattern *pattern)` [Function]

Initialize *pattern* to the empty pattern. This function belongs to `baz_prolongation_pattern.c`. It is run by 105 tests, including 6 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_reset_prolongation_pattern (struct baz_prolongation_pattern *pattern)` [Function]

Reset *pattern* to the empty pattern. This function belongs to `baz_prolongation_pattern.c`. It is run by 105 tests, including 6 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`struct baz_prolongation_pattern * baz_new_prolongation_pattern (void)` [Function]

Allocate a new prolongation pattern, initialize it and return it. This function belongs to `baz_prolongation_pattern.c`. It is not run by any test.

`void baz_set_prolongation_pattern (struct baz_prolongation_pattern *dst, struct baz_prolongation_pattern *src)` [Function]

This function belongs to `baz_prolongation_pattern.c`. It is run by 100 tests, including 2 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`void baz_variable_mapping_prolongation_pattern (struct bav_tableof_variable *dst, struct bav_tableof_variable *src, struct baz_prolongation_pattern *pattern)` [Function]

Assign to *dst* all the variables obtained by evaluating *src* using *pattern*. This function belongs to `baz_prolongation_pattern.c`. It is run by 99 tests, including 2 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p baz_sizeof_prolongation_pattern (struct baz_prolongation_pattern *P, enum ba0_garbage_code code)` [Function]

Return the size of the memory needed to perform a copy of P. If *code* is `ba0_embedded` then P is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `baz_prolongation_pattern.c`. It is run by 11 tests, including 0 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`void baz_switch_ring_prolongation_pattern (struct baz_prolongation_pattern *P, struct bav_differential_ring *R)` [Function]

This function belongs to `baz_prolongation_pattern.c`. It is run by 11 tests, including 0 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`void * baz_scanf_prolongation_pattern (void *P)` [Function]

The general parser for prolongation patterns. It can be called through `ba0_scanf/%prolongation_pattern`. The expected syntax is described by the following example:

```
{
y[(x,k),(t,l)]:'y[k,l]/(factorial(k)*factorial(l))',
z[(x,k),(t,l)]:'factorial(k)*z[k]'
}
```

Observe that the identifiers k and l associated to the two derivations must be the same for each dependent variable. This function belongs to `baz_prolongation_pattern.c`. It is run by 105 tests, including 6 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_printf_prolongation_pattern (void *P)` [Function]
The general printing function for prolongation patterns. It can be called through `ba0_printf/%prolongation_pattern`. This function belongs to `baz_prolongation_pattern.c`. It is run by 45 tests, including 4 in the `baz` library and 0 in the `DifferentialAlgebra` package.

7.5 Series Mappings

Description of type `baz_series_mapping`. This data type permits to associate derivatives such as `y`, `y[x]`, `z[x,x]` to subscripted variables such as `y[0]`, `y[1]`, `z[2]`.

The field `Y` contains the table of the differential indeterminates concerned by the mapping. This table is sorted by increasing order with respect to the ranking.

The field `dict_Y` maps any differential indeterminate y to its index in `Y`.

The field `Ybar` contains the prolongation pattern which permits to evaluate derivatives of any y in `Y` to subscripted variables, possibly multiplied by some coefficient. Assume for instance that the second derivative derivative $\ddot{y}$ gets evaluated to y_2. Then the map defined by `Ybar` associates the subscripted variable `y[2]` to the variable `y[x,x]`.

Let i be the index of some y in `Y`. Then `R.tab[i]` contains the value of the `index_in_rigs` field of the subscripted variables associated to y. Indeed: all the subscripted variables generated from a single y in `Y` have different symbols but all these symbols share the same `index_in_rigs` field.

The field `dict_R` maps any `index_in_rigs` to its index in `R` hence to the index of the corresponding differential indeterminate in `Y`.

Let i be the index of some y in `Y`. Then `ozs.tab[i]` contains the subscript associated (by `Ybar`) to the order zero derivative of y. Its default value is 0. It is equal to -1 if y has no defining ODE.

The field `x` contains the independent variable. This type declaration belongs to `baz_series_mapping.h`.

```
struct baz_series_mapping
{
  struct bav_tableof_symbol Y;
  struct bav_dictionary_symbol dict_Y;
  struct baz_prolongation_pattern Ybar;
  struct ba0_tableof_int_p R;
  struct ba0_dictionary_int_p dict_R;
  struct ba0_tableof_int_p ozs;
  struct bav_symbol *x;
};
```

`void baz_init_series_mapping (struct baz_series_mapping *M)` [Function]
Initialize M to the empty series mapping. This function belongs to `baz_series_mapping.c`. It is run by 99 tests, including 2 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`struct baz_series_mapping * baz_new_series_mapping (void)` [Function]
Allocate a new series mapping, initialize it and return it. This function belongs to `baz_series_mapping.c`. It is not run by any test.

`void baz_set_series_mapping (struct baz_series_mapping *dst, struct baz_series_mapping *src)` [Function]

Assign *src* to *dst*. This function belongs to `baz_series_mapping.c`. It is not run by any test.

`void baz_set_spx_series_mapping (struct baz_series_mapping *M, struct bav_tableof_symbol *Y, struct baz_prolongation_pattern *Ybar, struct bav_symbol *x)` [Function]

Assign to M the series mapping defined by Y, $Ybar$ and x. Exceptions `BAZ_ERRPAT` and `BAZ_ERRIND` may be raised. This function belongs to `baz_series_mapping.c`. It is run by 99 tests, including 2 in the `baz` library and 10 in the `DifferentialAlgebra` package.

`struct bav_variable * baz_derivative_to_subscripted (struct bav_variable *derivative, struct baz_series_mapping *M)` [Function]

Return the subscripted variable associated to *derivative* with respect to M. Return `BAV_NOT_A_VARIABLE` if no such subscripted variable exists. This function belongs to `baz_series_mapping.c`. It is run by 3 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`struct bav_variable * baz_subscripted_to_derivative (struct bav_variable *subscripted, struct baz_series_mapping *M)` [Function]

Return the derivative associated to the subscripted variable *subscripted* with respect to M. Return `BAV_NOT_A_VARIABLE` if no such derivative exists. This function belongs to `baz_series_mapping.c`. It is run by 17 tests, including 1 in the `baz` library and 0 in the `DifferentialAlgebra` package.

7.6 Points with rational fraction values

Description of type `baz_value_ratfrac`. This data type permits to associate a `baz_ratfrac` value to a variable. This type declaration belongs to `baz_point_ratfrac.h`.

```
struct baz_value_ratfrac
{
  struct bav_variable *var;
  struct baz_ratfrac *value;
};
```

Description of type `baz_point_ratfrac`. This data type is a particular case of the type `struct ba0_point`. It permits to associate `baz_ratfrac` values to many different variables. Many functions assume the `tab` field to be sorted (see `ba0_sort_point`). They can be parsed using `ba0_scanf/%point(%Qz)` and printed by `ba0_printf/%point(%Qz)`. This type declaration belongs to `baz_point_ratfrac.h`.

```
struct baz_point_ratfrac
{
  ba0_int_p alloc;
  ba0_int_p size;
  struct baz_value_ratfrac **tab;
};
```

`void baz_init_value_ratfrac (struct baz_value_ratfrac *V)` [Function]
Initialize *V*. The `var` field is initialized at `BAV_NOT_A_VARIABLE`. This function belongs to `baz_point_ratfrac.c`. It is run by 103 tests, including 5 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`struct baz_value_ratfrac * baz_new_value_ratfrac (void)` [Function]
Allocate a new `struct baz_value_ratfrac`, initialize it and return it. This function belongs to `baz_point_ratfrac.c`. It is run by 103 tests, including 5 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_set_value_ratfrac (struct baz_value_ratfrac *dst, struct baz_value_ratfrac *src)` [Function]
Assign *src* to *dst*. This function belongs to `baz_point_ratfrac.c`. It is not run by any test.

`void baz_set_point_ratfrac (struct baz_point_ratfrac *dst, struct baz_point_ratfrac *src)` [Function]
Assign *src* to *dst*. This function belongs to `baz_point_ratfrac.c`. It is not run by any test.

`void baz_prolongate_point_ratfrac_variable (struct baz_point_ratfrac *R, struct baz_point_ratfrac *P, struct bav_variable *v)` [Function]
If *v* is a proper derivative of the `var` field of some value in *P*, extend *P* with new values obtained by differentiation (including one with `var` field equal to *v*). If *v* is not a proper derivative of the `var` field of some value in *P*, the point is left unchanged. The resulting point is stored in *R*. This function belongs to `baz_point_ratfrac.c`. It is run by 3 tests, including 3 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void baz_prolongate_point_ratfrac_term (struct baz_point_ratfrac *R, struct baz_point_ratfrac *P, struct bav_term *term)` [Function]
Apply `baz_prolongate_point_ratfrac_variable` to each variable occurring in *term*. Result in *R*. This function belongs to `baz_point_ratfrac.c`. It is run by 3 tests, including 3 in the `baz` library and 0 in the `DifferentialAlgebra` package.

`void baz_prolongate_point_ratfrac_using_pattern_variable (struct baz_point_ratfrac *R, struct baz_point_ratfrac *P, struct baz_prolongation_pattern *pattern, struct bav_variable *v)` [Function]
Assign to *R* the point obtained by adding to *P* one entry for *v* using *pattern*. If *v* does not match *pattern* then *R* is equal to *P*. This function belongs to `baz_point_ratfrac.c`. It is run by 101 tests, including 3 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_prolongate_point_ratfrac_using_pattern_term` [Function]
`(struct baz_point_ratfrac *R, struct baz_point_ratfrac *P, struct baz_prolongation_pattern *pattern, struct bav_term *term)`

Apply `baz_prolongate_point_ratfrac_using_pattern_variable` to each variable occurring in *term*. Result in *R*. This function belongs to `baz_point_ratfrac.c`. It is run by 61 tests, including 1 in the `baz` library and 11 in the `DifferentialAlgebra` package.

`void baz_eval_to_ratfrac_at_point_ratfrac_ratfrac (struct baz_ratfrac *R, struct baz_ratfrac *A, struct baz_point_ratfrac *point)` [Function]

Assign to *R* the rational fraction obtained by evaluating *A* at *point*. The evaluation is non differential. This function belongs to `baz_eval_ratfrac.c`. It is run by 14 tests, including 13 in the `baz` library and 1 in the `DifferentialAlgebra` package.

`void baz_evaluate_to_ratfrac_at_point_ratfrac_ratfrac` [Function]
`(struct baz_ratfrac *R, struct baz_ratfrac *A, struct baz_point_ratfrac *point)`

Assign to *R* the rational fraction obtained by evaluating *A* at *point*. The evaluation is differential: the evaluation point used for evaluating *A* is obtained by prolongating *point* (see `baz_prolongate_point_ratfrac`) but the parameter *point* is left unchanged. This function belongs to `baz_eval_ratfrac.c`. It is run by 3 tests, including 3 in the `baz` library and 0 in the `DifferentialAlgebra` package.

8 The bad library

8.1 Introduction

The `bad` library implements simplifiers for systems of differential polynomials with long integers `mpz_t` coefficients. The `bad` library relies on the `ba0`, `bav`, `bap` and `baz` libraries. The main implemented simplifier is the *Rosenfeld–Gröbner* algorithm (available in the `MAPLE` `diffalg` and `DifferentialAlgebra` packages). The related simplifier *pardi* is implemented too. No Gröbner basis computation is performed in this package. Regular chains (algebraic and differential) are implemented.

In this library, every polynomial is viewed as a polynomial equation if nothing is precised. Moreover, the only available polynomial type is a differential polynomial one `bap_polynom_mpz`. It is used to describe as well differential systems as nondifferential ones.

This documentation may be difficult to read for people not familiar with algebraic and differential simplification theory (i.e. everybody but one hundred persons). Such introductory texts can be found on the `DifferentialAlgebra` web site.

Read the documentation of the `ba0` library about settings.

`void bad_reset_all_settings (void)` [Function]

Reset all settings variables to their default values. This function must be called outside sequences of calls to the library. This function belongs to `bad_common.c`. It is run by 65 tests, including 0 in the `bad` library and 65 in the `DifferentialAlgebra` package.

8.1.1 Basic types, constants and naming conventions

The name of every exported object defined in the library starts by `bad_`. Functions starting by `bad_init` initialize raw data structures (constructors). Functions starting by `bad_new` dynamically allocate data structures and apply the corresponding `bad_init` functions on them. Functions starting by `bad_reset` reset data structures which are assumed to be already initialized. Functions starting by `bad_set` perform assignments and systematically copy the sources to the destinations.

A *sequence of calls* to the `bad` library starts with a call to `bad_restart` and finishes by a call to `bad_terminate`.

`void bad_restart (ba0_int_p` *`time_limit`*`, ba0_int_p` *`memory_limit`*`)` [Function]

Call the `baz_restart` function with the same parameters and define a few more formats. This function belongs to `bad_common.c`. It is run by 280 tests, including 124 in the `bad` library and 65 in the `DifferentialAlgebra` package.

`void bad_terminate (enum ba0_restart_level` *`level`*`)` [Function]

Call the `bap_terminate` function with the same parameter. This function belongs to `bad_common.c`. It is run by 280 tests, including 124 in the `bad` library and 65 in the `DifferentialAlgebra` package.

8.2 Global variables and settings

Setting variables are global configuration variables which permit to tune the behaviour of the library functions. This tuning is supposed to be performed before `bad_restart` is called (more generally, outside any sequence of calls the library). They are bundled in the `bad_initialized_global` global variable which has the following type.

Description of type `bad_global`. This type declaration belongs to `bad_global.h`.

```
struct bad_global
{
  struct
  {
/*
 * Local variable to bad_reduction.
 * Used to pass some extra information to a subfunction in
 * bad_random_eval_variables_under_the_stairs
 * Its value is meaningless between two calls to this function.
 */
    struct bav_tableof_variable *stairs;
  } reduction;
  struct
  {
/*
 * Statistical information set by bad_Rosenfeld_Groebner and bad_pardi
 */
    time_t begin;
    time_t end;
    ba0_int_p critical_pairs_processed;
    ba0_int_p reductions_to_zero;
  } stats;
};
```

Description of type `bad_initialized_global`. This type declaration belongs to `bad_global.h`.

```
struct bad_initialized_global
{
  struct
  {
/*
 * reduction_strategy = the type of reduction strategy applied
 * redzero_strategy   = the type of reduction test to zero applied
 * number_of_redzero_tries = tuning for probabilistic methods
 * Local to bad_reduction.
 */
    enum bad_typeof_reduction_strategy reduction_strategy;
    enum bad_typeof_redzero_strategy redzero_strategy;
    ba0_int_p number_of_redzero_tries;
  } reduction;
  struct
  {
/*
 * strategy = the type of regularization strategy applied
 * Local to bad_regularize.
 */
    enum bad_typeof_regularize_strategy strategy;
  } regularize;
  struct
  {
```

```
/*
 * The string used for denoting differential regular chain elements
 * in the context of preparation equations (cf. Low Power Theorem).
 */
    char *zstring;
  } preparation;
};
```

`void bad_init_stats (void)` [Function]

Reset to zero all counters (see the `bad_global.stats` variable). This function is called by `bad_pardi` and `bad_Rosenfeld_Groebner`. This function belongs to `bad_stats.c`. It is run by 156 tests, including 77 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_printf_stats (void *a)` [Function]

Print the statistics on the standard output. See the `bad_global.stats` variable. This function belongs to `bad_stats.c`. It is not run by any test.

8.3 Properties of regular chains

Description of type `bad_property_attchain`. This data type permits to describe the properties of regular chains. It is a subtype of `bad_attchain` This type declaration belongs to `bad_attchain.h`.

```
enum bad_property_attchain
{
// the ideal defined by the chain is differential
  bad_differential_ideal_property,
// the ideal defined by the chain is prime
  bad_prime_ideal_property,
// the chain is coherent (only relevant in the partial differential case)
  bad_coherence_property,
// the chain is autoreduced
  bad_autoreduced_property,
// the chain is squarefree
  bad_squarefree_property,
// the chain elements, viewed as univariate polynomials in their
// leaders, are primitive
  bad_primitive_property,
// the chain is strongly normalized
  bad_normalized_property
};
```

Description of type `bad_attchain`. This data type is a subtype of `bad_regchain`. It permits to define the *attributes* associated to a regular chain. This type declaration belongs to `bad_attchain.h`.

```
struct bad_attchain
{
// the ordering with respect to which the regular chain is defined
  bav_Iordering ordering;
// the properties bitwise encoded over an integer
  ba0_int_p property;
};
```

`void bad_init_attchain (struct bad_attchain *A)` [Function]
Set the field `ordering` to the current ordering. Clear the field `property` of A. This function belongs to `bad_attchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_reset_attchain (struct bad_attchain *A)` [Function]
Set the field `ordering` to the current ordering. Clear the field `property` of A. This function belongs to `bad_attchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_set_attchain (struct bad_attchain *A, struct bad_attchain *B)` [Function]
Assign B to A. This function belongs to `bad_attchain.c`. It is run by 232 tests, including 113 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_intersect_attchain (struct bad_attchain *A, struct bad_attchain *B)` [Function]
Intersect the attributes of A and those of B. The result is in A. This function is used to compute the attributes of the intersection of two regular chains (see `struct bad_intersectof_regchain *`). Exception `BAD_ERRIAC` is raised if A and B have different orderings or are not both differential or not both non differential. The prime ideal property of A is cleared. The other properties are intersected. This function belongs to `bad_attchain.c`. It is run by 7 tests, including 5 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_set_properties_attchain (struct bad_attchain *A, struct ba0_tableof_string *P)` [Function]
The list P contain a list of property identifiers. Set the corresponding properties in A. Exception `BAD_ERRNAC` is raised in the case of a non recognized property. This function belongs to `bad_attchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_set_automatic_properties_attchain (struct bad_attchain *A)` [Function]
Set some properties of A by applying the following rules. If *differential* is set then *squarefree* is set. If *prime* is set then *squarefree* is set. If *normalized* is set then *autoreduced* and *primitive* are set. This function belongs to `bad_attchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_set_property_attchain (struct bad_attchain *A, enum bad_property_attchain p)` [Function]
Set the property p of A. This function belongs to `bad_attchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_clear_property_attchain (struct bad_attchain *A, enum bad_property_attchain p)` [Function]
Clear the property p of A. This function belongs to `bad_attchain.c`. It is run by 126 tests, including 44 in the `bad` library and 31 in the `DifferentialAlgebra` package.

`bool bad_is_a_property_attchain (char *s, enum bad_property_attchain *p)` [Function]

Return `true` if s is the identifier of a property, `false` otherwise. Valid values for s are `autoreduced`, `primitive`, `squarefree`, `coherent` and `normalized`. This function belongs to `bad_attchain.c`. It is run by 235 tests, including 116 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_properties_attchain (struct ba0_tableof_string *T, struct bad_attchain *C)` [Function]

Append to T a list of identifiers describing the properties which are set in C. This function belongs to `bad_attchain.c`. It is run by 154 tests, including 86 in the `bad` library and 18 in the `DifferentialAlgebra` package.

`bool bad_has_property_attchain (struct bad_attchain *A, enum bad_property_attchain p)` [Function]

Return `true` if the property p is set in A, `false` otherwise. This function belongs to `bad_attchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`bool bad_defines_a_differential_ideal_attchain (struct bad_attchain *C)` [Function]

Return `true` if the differential ideal property is set in C, `false` otherwise. This function belongs to `bad_attchain.c`. It is run by 229 tests, including 114 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`bool bad_defines_a_prime_ideal_attchain (struct bad_attchain *C)` [Function]

Return `true` if the prime ideal property is set in C, `false` otherwise. This function belongs to `bad_attchain.c`. It is run by 200 tests, including 110 in the `bad` library and 31 in the `DifferentialAlgebra` package.

`bool bad_equal_attchain (struct bad_attchain *A, struct bad_attchain *B)` [Function]

Return `true` if A is equal to B, `false` otherwise. This function belongs to `bad_attchain.c`. It is not run by any test.

8.4 Regular chains

Description of type `bad_regchain`. This data type implements regular chains. Mathematically, a regular chain A defines an ideal which is either $(A) : I_A^\infty$ in the nondifferential case, or $[A] : H_A^\infty$ in the differential case.

The field `number` is used to associate a number to the regular chain, in order to identify it precisely in the *splitting trees* generated by elimination algorithms. If the regular chain is part of a *quadruple*, the number is used also to identify the quadruple.

The field `attrib` contains the *attributes* of the regular chain: the *ordering* with respect to which the chain is defined plus some properties.

The field `decision_system` contains the differential polynomials, sorted increasingly with respect to the chain ordering. This type declaration belongs to `bad_regchain.h`.

`struct bad_regchain`

```
{
// the number of the regular chain in a splitting tree
  ba0_int_p number;
// the attributes of the regular chain - including the ordering
  struct bad_attchain attrib;
// the polynomial set, sorted increasingly with respect to the chain ordering
  struct bap_tableof_polynom_mpz decision_system;
};
```

`void bad_init_regchain (struct bad_regchain *C)` [Function]
Initialize C to the empty chain. This function belongs to `bad_regchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_reset_regchain (struct bad_regchain *C)` [Function]
Reset C to the empty chain. This function belongs to `bad_regchain.c`. It is run by 97 tests, including 28 in the `bad` library and 24 in the `DifferentialAlgebra` package.

`struct bad_regchain * bad_new_regchain (void)` [Function]
Allocate a new regular chain, initialize it and return it. This function belongs to `bad_regchain.c`. It is run by 180 tests, including 96 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_realloc_regchain (struct bad_regchain *C, ba0_int_p n)` [Function]
Realloc the decision system of C so that it can receive at least n polynomials. Existing elements are preserved. This function belongs to `bad_regchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_set_regchain (struct bad_regchain *A, struct bad_regchain *B)` [Function]
Assign B to A. This function belongs to `bad_regchain.c`. It is run by 242 tests, including 123 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`ba0_int_p bad_get_number_regchain (struct bad_regchain *C)` [Function]
Return the value of the field `number` of C. This function belongs to `bad_regchain.c`. It is run by 145 tests, including 66 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_number_regchain (struct bad_regchain *C, ba0_int_p number)` [Function]
Assign *number* to the field `number` of C. This function belongs to `bad_regchain.c`. It is run by 154 tests, including 75 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_mark_indets_regchain (struct bav_dictionary_variable *dict, struct bav_tableof_variable *vars, struct bad_regchain *C)` [Function]
Append to *vars* the variables occurring in C which are not already present in *vars*. Every element of *vars* is supposed to be registered in the dictionary *dict*. This function belongs to `bad_regchain.c`. It is not run by any test.

`void bad_insert_polynom_in_regchain (struct bad_regchain *A, struct bap_polynom_mpz *p)` [Function]

Insert p in A. This low level function does not check if p can be inserted in A. This function belongs to `bad_regchain.c`. It is not run by any test.

`void bad_extend_regchain (struct bad_regchain *A, struct bad_regchain *B)` [Function]

Append the polynomials of B to A. If A is empty, assign the properties of B to A. Exception `BAD_ERRCRC` is raised if B is not compatible with A (see `bad_is_a_compatible_regchain`) or if the lowest leader of B is less than or equal to the highest leader of A. This function belongs to `bad_regchain.c`. It is run by 147 tests, including 68 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_regchain_tableof_polynom_mpz (struct bad_regchain *C, struct bap_tableof_polynom_mpz *T, struct ba0_tableof_string *properties, bool pretend)` [Function]

Assign to C the table T with properties given in *properties* plus some other ones (see `bad_set_automatic_properties_attchain`).

The table T does not need to be sorted. If the boolean *pretend* is `true`, then that's all.

Otherwise, the function checks that the polynomials depend on dependent variables and form a triangular set (and a differentially triangular one in the differential case). If they do not, exception `BAD_ERRNRC` is raised. Lastly, the function `bad_quench_regchain` is called to make the regular chain hold the properties with the exception of the coherence, prime ideal and differential ideal properties. This function may raise the exceptions `BAD_EXRCNC` or `BAD_EXRDDZ`. This function belongs to `bad_regchain.c`. It is run by 223 tests, including 121 in the `bad` library and 26 in the `DifferentialAlgebra` package.

`void bad_set_regchain_tableof_ratfrac_mpz (struct bad_regchain *C, struct baz_tableof_ratfrac *T, struct ba0_tableof_string *properties, bool pretend)` [Function]

Apply `bad_set_regchain_tableof_polynom_mpz` over the set of numerators of the rational fractions. See that function. This function belongs to `bad_regchain.c`. It is run by 218 tests, including 120 in the `bad` library and 22 in the `DifferentialAlgebra` package.

`void bad_set_property_regchain (struct bad_regchain *C, enum bad_property_attchain prop)` [Function]

Set property *prop* of C. This function belongs to `bad_regchain.c`. It is run by 229 tests, including 114 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_clear_property_regchain (struct bad_regchain *C, enum bad_property_attchain prop)` [Function]

Clear property *prop* of C. This function belongs to `bad_regchain.c`. It is run by 9 tests, including 9 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_set_properties_regchain (struct bad_regchain *C, struct ba0_tableof_string *T)` [Function]

Set the properties of C with T. Do not set properties automatically (see `bad_set_automatic_properties_attchain`). This function belongs to `bad_regchain.c`. It is run by 59 tests, including 8 in the `bad` library and 12 in the `DifferentialAlgebra` package.

`void bad_set_automatic_properties_regchain (struct bad_regchain *C)` [Function]

Set the automatic properties of C. See `bad_set_automatic_properties_attchain`. This function belongs to `bad_regchain.c`. It is run by 49 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`bool bad_has_property_regchain (struct bad_regchain *C, enum bad_property_attchain P)` [Function]

Return `true` if C has the property P. This function belongs to `bad_regchain.c`. It is run by 230 tests, including 113 in the `bad` library and 34 in the `DifferentialAlgebra` package.

`void bad_properties_regchain (struct ba0_tableof_string *properties, struct bad_regchain *C)` [Function]

Assign to *properties* the properties of C. This function belongs to `bad_regchain.c`. It is run by 59 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`bool bad_defines_a_differential_ideal_regchain (struct bad_regchain *C)` [Function]

Return `true` if C holds the `bad_differential_ideal_property` else `false`. This function belongs to `bad_regchain.c`. It is run by 229 tests, including 114 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`bool bad_defines_a_prime_ideal_regchain (struct bad_regchain *C)` [Function]

Return `true` if C holds the `bad_prime_ideal_property` else `false`. This function belongs to `bad_regchain.c`. It is run by 188 tests, including 98 in the `bad` library and 31 in the `DifferentialAlgebra` package.

`ba0_int_p bad_codimension_regchain (struct bad_regchain *C, struct bad_base_field *K)` [Function]

Return the codimension (differential or not, depending if C holds the `bad_differential_ideal_property`) of the ideal defined by C over the field K.

Differential polynomials are assumed to have coefficients in K. In the algebraic case, the codimension is just the number of equations of C which do not belong to K.

In the ordinary case, the codimension is the number of equations of C which do not belong to K plus the number of parameters involved in these equations which are not leaders (we have to count the implicit equations which state that the derivatives of the parameters are zero).

In the partial differential case, the formula is the same as in the ordinary case but differential polynomials whose leaders are derivatives of the same differential indeterminate are counted only once. This function belongs to `bad_regchain.c`. It is run by 22 tests, including 17 in the `bad` library and 5 in the `DifferentialAlgebra` package.

`void bad_sort_regchain (struct bad_regchain *A, struct bad_regchain *B)` [Function]

Assign to A the regular chain obtained by sorting the polynomials of B with respect to the current ordering. The elements of A are readonly. Properties of B become properties of A. This function belongs to `bad_regchain.c`. It is run by 97 tests, including 28 in the `bad` library and 24 in the `DifferentialAlgebra` package.

`bool bad_is_rank_of_regchain (struct bav_rank *rg, struct bad_regchain *C, ba0_int_p *i)` [Function]

Return `true` if *rg* is the rank of some element of C. If so and *i* is not the zero pointer, *i* receives the index of the element. This function belongs to `bad_regchain.c`. It is run by 26 tests, including 13 in the `bad` library and 13 in the `DifferentialAlgebra` package.

`bool bad_is_leader_of_regchain (struct bav_variable *v, struct bad_regchain *C, ba0_int_p *i)` [Function]

Return `true` if v is the leading derivative of some element of C. If so and *i* is not the zero pointer, $*i$ receives the index of the element. This function belongs to `bad_regchain.c`. It is run by 217 tests, including 110 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`bool bad_depends_on_leader_of_regchain (struct bap_polynom_mpz *A, struct bad_regchain *C)` [Function]

Return `true` if A depends on the leader of some element of C, else `false`. This function belongs to `bad_regchain.c`. It is run by 223 tests, including 108 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_leaders_of_regchain (struct bav_tableof_variable *leaders, struct bad_regchain *C)` [Function]

Store in *leaders* the leaders of the `decision_system` field of A. This function belongs to `bad_regchain.c`. It is run by 53 tests, including 4 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`bool bad_is_derivative_of_leader_of_regchain (struct bav_variable *v, struct bad_regchain *C, ba0_int_p *i)` [Function]

Return `true` if v is a derivative of the leading derivative of some element of C. If so and *i* is not the zero pointer, $*i$ receives the index of the element. This function belongs to `bad_regchain.c`. It is run by 226 tests, including 109 in the `bad` library and 34 in the `DifferentialAlgebra` package.

`bool bad_is_solved_regchain (struct bad_regchain *C)` [Function]

Return `true` if C could be written as $p(x_1) = 0$, $x_2 = f_2(x_1)$, $\ldots$, $x_n = f_n(x_1)$ where p and the f_i are polynomials. This function belongs to `bad_regchain.c`. It is not run by any test.

`bool bad_is_orthonomic_regchain (struct bad_regchain *C, struct bad_base_field *K)` [Function]

Return `true` if the elements of C have leading degrees 1 and initials in K. This function belongs to `bad_regchain.c`. It is not run by any test.

`bool bad_is_explicit_regchain (struct bad_regchain *C)` [Function]

Return `true` if the elements of C have leading degrees 1. This function belongs to `bad_regchain.c`. It is run by 146 tests, including 62 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`bool bad_is_a_compatible_regchain (struct bad_regchain *C, struct bad_attchain *attrib)` [Function]

Return `true` if C can be included as is in a regular chain having properties *attrib*. The orderings must be the same and every property of *attrib* must be a property of C. This function belongs to `bad_regchain.c`. It is run by 147 tests, including 68 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_fast_primality_test_regchain (struct bad_regchain *C)` [Function]

Set the `bad_prime_ideal_property` of C, if C is proven (after a fast test) to define a prime ideal. This function belongs to `bad_regchain.c`. It is run by 146 tests, including 62 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`bool bad_is_zero_regchain (struct bad_regchain *C)` [Function]

Return `true` if C is empty. This function belongs to `bad_regchain.c`. It is run by 233 tests, including 114 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`bav_Iordering bad_ordering_eliminating_leaders_of_regchain (struct bad_regchain *C)` [Function]

Return an ordering r with respect to which the leaders of the elements of C are greater than any other variable. The ordering is not necessarily a ranking in the sense of Kolchin. The ordering between the leaders on the one hand and and other variables on the other hand are preserved. This function does not change the current ordering. This function belongs to `bad_regchain.c`. It is run by 97 tests, including 28 in the `bad` library and 24 in the `DifferentialAlgebra` package.

`ba0_int_p bad_product_of_leading_degrees_regchain (struct bad_regchain *C)` [Function]

Return the product of the leading degrees of the elements of C. This function belongs to `bad_regchain.c`. It is not run by any test.

Description of type `bad_typeof_inclusion_test_result`. This data type provides a return code after an inclusion test between ideals presented by regular chains. This type declaration belongs to `bad_reduction.h`.

```
enum bad_typeof_inclusion_test_result
{
  bad_inclusion_test_positive,
  bad_inclusion_test_negative,
  bad_inclusion_test_uncertain
};
```

`enum bad_typeof_inclusion_test_result` [Function]
`bad_is_included_regchain (struct bad_regchain *A, struct bad_regchain *B, struct bad_base_field *K)`
Test if A is included in B. Polynomials are supposed to have coefficients in K. This function belongs to `bad_reduction.c`. It is run by 28 tests, including 22 in the `bad` library and 6 in the `DifferentialAlgebra` package.

`void bad_inequations_regchain (struct bap_tableof_polynom_mpz *res, struct bad_regchain *C)` [Function]
Assign to *res* the initials and separants of C. Numeric polynomials are discarded. Polynomials are made primitive. Redundant polynomials are discarded. Some polynomials may be factored by means of `baz_factor_easy_polynom_mpz`. This function belongs to `bad_regchain.c`. It is not run by any test.

`void * bad_scanf_regchain (void *A)` [Function]
A parsing function for regular chains. It is called by `ba0_scanf/%regchain`. The parsed string is expected to have the form `regchain ([rational fractions], [properties])`. The input list of rational fractions needs not be sorted. The function checks that the numerators of *ratfrac* form a regular chain. It calls `bad_quench_regchain` to make the chain satisfy the properties listed in *property* except the coherence, prime ideal and differential ideal properties. This function belongs to `bad_regchain.c`. It is run by 170 tests, including 94 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void * bad_scanf_pretend_regchain (void *A)` [Function]
A parsing function for regular chains. It is called by `ba0_scanf/%pretend_regchain`. The parsed string is expected to have the form `regchain ([rational fractions], [properties])`. The input list of rational fractions needs not be sorted. The regular chain property of the resulting list is not checked. This function belongs to `bad_regchain.c`. It is run by 65 tests, including 43 in the `bad` library and 22 in the `DifferentialAlgebra` package.

`void bad_printf_regchain (void *A)` [Function]
A printing function for regular chains. It is called by `ba0_printf/%regchain`. The output format is `regchain ([rational fractions], [properties])`. This function belongs to `bad_regchain.c`. It is run by 144 tests, including 101 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_printf_regchain_equations (void *A)` [Function]
A printing function for regular chains. It is called by `ba0_printf/%regchain_equations`. The output format is the one of a table of polynomials. This function belongs to `bad_regchain.c`. It is run by 67 tests, including 64 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bad_sizeof_regchain (struct bad_regchain *C, enum ba0_garbage_code code)` [Function]
Return the size of the memory needed to perform a copy of C. If *code* is `ba0_embedded` then C is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `bad_regchain.c`. It is run by 37 tests, including 0 in the `bad` library and 36 in the `DifferentialAlgebra` package.

`void bad_switch_ring_regchain (struct bad_regchain *C, struct bav_differential_ring *R)` [Function]

This low level function should be used in conjunction with `bav_set_differential_ring`: if R is a ring obtained by application of `bav_set_differential_ring` to the ring C refers to, then this function makes C refer to R. The chain C is modified. This function belongs to `bad_regchain.c`. It is run by 37 tests, including 0 in the `bad` library and 36 in the `DifferentialAlgebra` package.

8.5 Intersections of regular chains

Description of type `bad_intersectof_regchain`. This data structure mostly is a table a regular chain. Mathematically, the ideal that it represents is the intersection of the ideals defined by the chains. This type declaration belongs to `bad_intersectof_regchain.h`.

```
struct bad_intersectof_regchain
{
  struct bad_attchain attrib;
  struct bad_tableof_regchain inter;
};
```

`void bad_init_intersectof_regchain (struct bad_intersectof_regchain *ideal)` [Function]

Initialize *ideal* to the empty intersection. This function belongs to `bad_intersectof_regchain.c`. It is run by 179 tests, including 100 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_reset_intersectof_regchain (struct bad_intersectof_regchain *ideal)` [Function]

Reset *ideal* to the empty intersection. This function belongs to `bad_intersectof_regchain.c`. It is not run by any test.

`struct bad_intersectof_regchain * bad_new_intersectof_regchain (void)` [Function]

Allocate a new intersection, initialize it and return it. This function belongs to `bad_intersectof_regchain.c`. It is run by 152 tests, including 73 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_realloc_intersectof_regchain (struct bad_intersectof_regchain *ideal, ba0_int_p n)` [Function]

Realloc the table `inter` of *ideal* if needed in such a way that the table can receive at least n regular chains. Formerly stored regular chains are kept. This function belongs to `bad_intersectof_regchain.c`. It is run by 172 tests, including 94 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_intersectof_regchain (struct bad_intersectof_regchain *I, struct bad_intersectof_regchain *J)` [Function]

Assign J to I. This function belongs to `bad_intersectof_regchain.c`. It is run by 172 tests, including 94 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_intersectof_regchain_regchain (struct bad_intersectof_regchain *ideal, struct bad_regchain *C)` [Function]

Assign C to *ideal*. This function belongs to `bad_intersectof_regchain.c`. It is run by 172 tests, including 94 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_append_intersectof_regchain_regchain (struct bad_intersectof_regchain *ideal, struct bad_regchain *C)` [Function]

Append the regular chain C to *ideal*. If *ideal* is empty, assign the attributes of C to those of *ideal* else intersects the attributes of C with those of *ideal* by calling `bad_intersect_attchain`. Exception `BAD_ERRIAC` may be raised. This function belongs to `bad_intersectof_regchain.c`. It is run by 10 tests, including 8 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_append_intersectof_regchain (struct bad_intersectof_regchain *I, struct bad_intersectof_regchain *J)` [Function]

Append the regular chains of J to I. This function belongs to `bad_intersectof_regchain.c`. It is run by 10 tests, including 8 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_set_property_intersectof_regchain (struct bad_intersectof_regchain *I, enum bad_property_attchain P)` [Function]

Set the property P of I. This function belongs to `bad_intersectof_regchain.c`. It is not run by any test.

`void bad_clear_property_intersectof_regchain (struct bad_intersectof_regchain *I, enum bad_property_attchain P)` [Function]

Clear the property P of I. This function belongs to `bad_intersectof_regchain.c`. It is run by 59 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_set_properties_intersectof_regchain (struct bad_intersectof_regchain *I, struct ba0_tableof_string *P)` [Function]

Set the properties of I with T. This function belongs to `bad_intersectof_regchain.c`. It is run by 59 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_set_automatic_properties_intersectof_regchain (struct bad_intersectof_regchain *I)` [Function]

Set the automatic properties of I. See `bad_set_automatic_properties_attchain`. This function belongs to `bad_intersectof_regchain.c`. It is run by 59 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`bool bad_has_property_intersectof_regchain (struct bad_intersectof_regchain *I, enum bad_property_attchain P)` [Function]

Return `true` if I has the property P. This function belongs to `bad_intersectof_regchain.c`. It is run by 49 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_properties_intersectof_regchain (struct ba0_tableof_string *properties, struct bad_intersectof_regchain *I)` [Function]

Assign to *properties* the properties of *I*. This function belongs to `bad_intersectof_regchain.c`. It is not run by any test.

`struct bad_regchain * bad_get_regchain_intersectof_regchain (struct bad_intersectof_regchain *I, ba0_int_p number)` [Function]

Return the component of *I* whose `number` field is equal to *number*. Return zero if not found. This function belongs to `bad_intersectof_regchain.c`. It is run by 49 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_fast_primality_test_intersectof_regchain (struct bad_intersectof_regchain *ideal)` [Function]

Apply `bad_fast_primality_test_regchain` over each component of *ideal*. This function belongs to `bad_intersectof_regchain.c`. It is run by 147 tests, including 68 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_sort_intersectof_regchain (struct bad_intersectof_regchain *I, struct bad_intersectof_regchain *J)` [Function]

Sort the regular chains of *J* by decreasing order. The result is stored in *I*. Regular chains are compared following the classical ordering on differentially triangular sets: ranks of polynomials are compared from bottom up and, in case no difference of ranks is observed, the longer set is considered as lower than the shorter one. This function belongs to `bad_intersectof_regchain.c`. It is run by 64 tests, including 44 in the `bad` library and 20 in the `DifferentialAlgebra` package.

`void bad_remove_redundant_components_intersectof_regchain (struct bad_intersectof_regchain *I, struct bad_intersectof_regchain *J, struct bad_base_field *K)` [Function]

Perform the inclusion test between the components of *J*. For each regular chain *C*, if there exists another regular chain *D* such that the inclusion test $D \subset C$ is positive, then *C* is removed from the intersection.

This function relies on `bad_is_included_regchain` which may fail to decide if a regchain is included in another one. The computed intersection is thus not guaranteed to be minimal. Polynomials are supposed to have coefficients in *K*.

The resulting intersection is sorted (see `bad_sort_intersectof_regchain`). This function belongs to `bad_intersectof_regchain.c`. It is run by 64 tests, including 44 in the `bad` library and 20 in the `DifferentialAlgebra` package.

`void * bad_scanf_intersectof_regchain (void *I)` [Function]

A parsing function for intersections of regular chains. It is called by `ba0_scanf/%intersectof_regchain`. It relies on `bad_scanf_regchain`. This function belongs to `bad_intersectof_regchain.c`. It is run by 111 tests, including 91 in the `bad` library and 20 in the `DifferentialAlgebra` package.

`void * bad_scanf_intersectof_pretend_regchain (void *I)` [Function]
A parsing function for intersections of regular chains. It is called by `ba0_scanf/%intersectof_pretend_regchain`. It relies on `bad_scanf_pretend_regchain`. This function belongs to `bad_intersectof_regchain.c`. It is not run by any test.

`void bad_printf_intersectof_regchain (void *I)` [Function]
A printing function for intersections of regular chains. It is called by `ba0_printf/%intersectof_regchain`. This function belongs to `bad_intersectof_regchain.c`. It is run by 90 tests, including 90 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_printf_intersectof_regchain_equations (void *I)` [Function]
A printing function for intersections of regular chains. It is called by `ba0_printf/%intersectof_regchain_equations`. It relies on `bad_printf_regchain_equations`. This function belongs to `bad_intersectof_regchain.c`. It is run by 62 tests, including 62 in the `bad` library and 0 in the `DifferentialAlgebra` package.

8.6 Quenching regular chains

The quenching process is an important process which permits to transform triangular sets into intersections of regular chains. These triangular sets amy be given by the user or generated by algorithms when modifying a regular chain.

Description of type `bad_quench_map`. This data type describes the properties currently satisfied by a triangular set. It drives the process carried out by `bad_quench_regchain` and is updated after each elementary operation performed over the triangular set. Each field is associated to a property of the triangular set under consideration.

A property can be *inactive* (if not desired by the set) or *active*. In this second case, it can be satisfied by some elements of the triangular set or not.

Properties are encoded by a table of booleans. For a given property `prop`,

- if the field `prop.size` is zero then the property is inactive ;
- if it is nonzero then `prop.tab[k]` is `true` if and only if the property holds for the kth element of the triangular set.

This type declaration belongs to `bad_quench_map.h`.

```
struct bad_quench_map
{
// the number of elements of the triangular set being processed
  ba0_int_p nb_max_elem;
// the triangular set is partially autoreduced
  struct ba0_tableof_int_p partially_autoreduced;
// the triangular set is a regular chain
  struct ba0_tableof_int_p regular;
// the triangular set elements satisfy bad_primitive_property
  struct ba0_tableof_int_p primitive;
// the triangular set satisfies bad_autoreduced_property
  struct ba0_tableof_int_p autoreduced;
// the triangular set satisfies bad_squarefree_property
  struct ba0_tableof_int_p squarefree;
```

```
// the triangular set satisfies bad_normalized_property
  struct ba0_tableof_int_p normalized;
};
```

`void bad_init_quench_map (struct bad_quench_map *map, struct bad_regchain *A)` [Function]

Initialize *map*. The properties of A are assumed not to hold. This function belongs to `bad_quench_map.c`. It is run by 228 tests, including 113 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_init_from_complete_quench_map (struct bad_quench_map *map, ba0_int_p k, struct bad_regchain *A)` [Function]

Initialize *map* in the context of the `bad_complete_quadruple` context (this function being called by `bad_Rosenfeld_Groebner` or `bad_pardi`).

The triangular set A has been obtained by inserting at index k, one new polynomial, in a regular chain satisfying all the properties as A (except `bad_normalized_property`).

This new polynomial is primitive and fully reduced w.r.t A.

The `bad_normalized_property` is inactivated. This function belongs to `bad_quench_map.c`. It is run by 148 tests, including 73 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_init_set_quench_map (struct bad_quench_map *dst, struct bad_quench_map *src)` [Function]

Assign *src* to *dst*. This function belongs to `bad_quench_map.c`. It is run by 8 tests, including 7 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_set_all_properties_quench_map (struct bad_quench_map *map, bool b)` [Function]

Declare all active properties of *map* to hold (if b is `true`) or not to hold (if it is `false`) for each element of the triangular set under consideration. This function belongs to `bad_quench_map.c`. It is run by 83 tests, including 53 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_property_quench_map (struct ba0_tableof_int_p *prop, bool b)` [Function]

If *prop* is active, declare this property to hold (if b is `true`) or not to hold (if it is `false`) for each element of the triangular set under consideration. This function belongs to `bad_quench_map.c`. It is run by 83 tests, including 53 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_inactivate_property_quench_map (struct ba0_tableof_int_p *prop)` [Function]

Change the status of property *prop* to inactive. This function belongs to `bad_quench_map.c`. It is run by 7 tests, including 7 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`ba0_int_p bad_first_index_quench_map (struct bad_quench_map *map)` [Function]

Return the smallest index k such that some property is not satisfied at this index. This function belongs to `bad_quench_map.c`. It is run by 228 tests, including 113 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_fully_reduced_polynom_quench_map (struct bad_quench_map *map, ba0_int_p k)` [Function]

Update *map* in the following context. The polynomial at index k has been replaced by another polynomial, with the same rank, obtained by full reduction. This function belongs to `bad_quench_map.c`. It is run by 77 tests, including 38 in the `bad` library and 17 in the `DifferentialAlgebra` package.

`void bad_partially_reduced_polynom_quench_map (struct bad_quench_map *map, ba0_int_p k)` [Function]

Update *map* in the following context. The polynomial at index k has been replaced by another polynomial, with the same rank, obtained by partial reduction. This function belongs to `bad_quench_map.c`. It is run by 1 tests, including 1 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_is_an_already_satisfied_property_quench_map (struct ba0_tableof_int_p *prop, ba0_int_p k)` [Function]

Update *prop* in the following context. The polynomial at index k is left unchanged because it is already satisfying the property. The dependencies between properties (autoreduced implies partially autoreduced, normalized implies regular) are not carried out by this function. This function belongs to `bad_quench_map.c`. It is run by 223 tests, including 108 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`bool bad_address_property_quench_map (struct ba0_tableof_int_p *prop, ba0_int_p k)` [Function]

Return `true` if property *prop* is satisfied for all indices strictly less than k but fails to hold at index k. This function belongs to `bad_quench_map.c`. It is run by 223 tests, including 108 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_primitive_polynom_quench_map (struct bad_quench_map *map, ba0_int_p k)` [Function]

Update *map* in the following context. The polynomial at index k has been replaced by its primitive part. This function belongs to `bad_quench_map.c`. It is run by 180 tests, including 94 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_normalized_polynom_quench_map (struct bad_quench_map *map, ba0_int_p k)` [Function]

Update *map* in the following context. The polynomial at index k has been replaced by another polynomial, with the same rank and a normalized leading coefficient, obtained by means of a multiplication by an algebraic inverse followed by a reduction. This function belongs to `bad_quench_map.c`. It is run by 25 tests, including 12 in the `bad` library and 13 in the `DifferentialAlgebra` package.

`void bad_pseudo_divided_polynom_quench_map (struct bad_quench_map *map, ba0_int_p k)` [Function]

Update *map* in the following context. The polynomial at index k has been replaced by its pseudo-quotient or pseudo-remainder computed with respect to a polynomial with a regular initial. The new polynomial has the same leader as the old one but a lower degree. This function belongs to `bad_quench_map.c`. It is run by 33 tests, including 28 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_printf_quench_map (struct bad_quench_map *map)` [Function]

Print *map* on the standard output. For debugging purposes. This function belongs to `bad_quench_map.c`. It is not run by any test.

`void bad_quench_regchain (struct bad_regchain *Cbar, struct bad_quench_map *map, struct bav_tableof_term *theta, bool *discarded_branch, struct bad_regchain *C, struct bad_base_field *K, struct bap_polynom_mpz * *ddz)` [Function]

This function transforms the triangular set C into one regular chain *Cbar*. The properties that *Cbar* must satisfy and are not satisfied by C are described in *map*. The process may fail and raise an exception.

If nonzero, the argument *discarded_branch* is set to `true` if one branch has been discarded during the computation (case of a common factor of an element of C and an initial or a separant). Otherwise, it is set to `false`.

Exception `BAD_EXRCNC` is raised if C is proved to be inconsistent.

Exception `BAD_EXRDDZ` is raised if a zerodivisor is exhibited. In this case, if *ddz* is nonzero, then it is assigned the exihibited zerodivisor, which has the same leader as some element of C and an initial regular with respect to C.

Polynomials are supposed to have coefficients in K. The `bad_coherent_property` is not handled in this function. This function belongs to `bad_quench_regchain.c`. It is run by 228 tests, including 113 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_quench_and_handle_exceptions_regchain (struct bad_intersectof_regchain *tabC, struct bad_quench_map *map, struct bav_tableof_term *theta, bool *discarded_branch, struct bad_regchain *ideal, struct bad_base_field *K)` [Function]

Apply `bad_quench_regchain` to C, *map* and K, where C denotes the last element of *tabC*, and handle the exceptions raised by this process.

The optional argument *ideal* is used to avoid splitting cases in the case of the `bad_pardi` algorithm.

The result is a possibly empty set of regular chains which are stored in *tabC* at indices greater than or equal to that of C. Exceptions are handled as follows.

If the exception `BAD_EXRCNC` is raised then the current chain is discarded.

If the exception `BAD_EXRDDZ` is raised then a factorization $g\,h$ of some $p_i \in C$ is exhibited. The chain C is split by replacing p_i either by g or by h and the algorithm is recursively applied over each branch.

If *ideal* is nonzero and g (resp. h) does not lie in the ideal that it defines, then the branch obtained by replacing p_i by g (resp. h) is discarded.

If nonzero, the argument *discarded_branch* is set to `true` if at least one branch is discarded. Otherwise it is set to `false`.

The `bad_coherent_property` is not handled by this function. This function belongs to `bad_quench_regchain.c`. It is run by 162 tests, including 87 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_handle_splitting_exceptions_regchain (struct bad_intersectof_regchain *tabC, struct bad_quench_map *map, struct bav_tableof_term *theta, bool *discarded_branch, struct bap_polynom_mpz *g, struct bad_regchain *ideal, char *raised, struct bad_base_field *K)` [Function]

Denote C the last element of *tabC*. While quenching C (see above) or performing a normal form computation modulo C, one of the exceptions `BAD_EXRCNC` and `BAD_EXRDDZ` can be raised. This function handles such exceptions, which are given in *raised*.

If the exception is `BAD_EXRCNC`, the regular chain C is just discarded from *tabC*.

If the exception is `BAD_EXRDDZ` then g contains a polynomial with the same leader as some $p_i \in C$ and an initial which is regular with respect to C. This polynomial provides a factorization $g\,h$ of $p_i \in C$.

The chain C is then split by replacing p_i either by g or by h. If *map* is zero, it is initialized assuming all properties of C are satisfied. It is then updated according to the exception and `bad_quench_and_handle_exceptions_regchain` is applied over each branch.

If *ideal* is nonzero and g (resp. h) does not lie in the ideal that it defines, then the branch obtained by replacing p_i by g (resp. h) is discarded.

Note: Exception `BAD_EXRNUL`, which looks pretty much like `BAD_EXRCNC` and may be raised while performing an algebraic inverse computation, is not handled in this function in order to allow a specific treatment by the calling function (normal form computations).

If nonzero, the argument *discarded_branch* is set to `true` if at least one branch is discarded. Otherwise it is set to `false`. This function belongs to `bad_quench_regchain.c`. It is run by 41 tests, including 20 in the `bad` library and 14 in the `DifferentialAlgebra` package.

8.7 Base fields

Description of type `bad_base_field`. This data structure implements base fields for polynomials, regular chains. They are used by elimination methods. Base fields are presented by generators and relations.

The field `first_block_index` contains the index of the first block which contains generators, in the table of blocks of the current ordering. The rule is that any dependent variable which belongs to the same block or to any lower block belongs to the base field. If no generator has been specified, the base field is defined as the field of the rational fractions in the independent variables and the field `first_block_index` contains `BAO_NOT_AN_INDEX`.

Base fields are seen to be defined with respect to some ordering. There is a restriction on orderings with respect to which base fields can be defined: they must not involve `varmax` variables.

A base field is differential if its `relations` field holds the `bad_differential_ideal_property`. This type declaration belongs to `bad_base_field.h`.

```
struct bad_base_field
{
// indicate if the polynomials which are going to be tested
// zero or nonzero may supposed to be reduced with respect to relations
  bool assume_reduced;
// the first block index containing generators
  ba0_int_p first_block_index;
// the base field defining equations - the corresponding ideal must be prime.
  struct bad_regchain relations;
};
```

`void bad_init_base_field (struct bad_base_field *K)` [Function]

Initialize the base field K to the field of the rational fractions in the independent variables. This function belongs to `bad_base_field.c`. It is run by 229 tests, including 114 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`struct bad_base_field * bad_new_base_field (void)` [Function]

Allocate a base field, initialize it and return it. This function belongs to `bad_base_field.c`. It is run by 79 tests, including 59 in the `bad` library and 20 in the `DifferentialAlgebra` package.

`bool bad_is_differential_base_field (struct bad_base_field *K)` [Function]

Return `true` if K is a differential field else `false`. This function belongs to `bad_base_field.c`. It is run by 82 tests, including 72 in the `bad` library and 6 in the `DifferentialAlgebra` package.

`void bad_set_base_field (struct bad_base_field *dst, struct bad_base_field *src)` [Function]

Assign *src* to *dst*. This function belongs to `bad_base_field.c`. It is run by 229 tests, including 114 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_base_field_generators (struct ba0_tableof_range_indexed_group *G, struct bad_base_field *K)` [Function]

Assign to G the identifiers of the differential indeterminates with plain symbols as well as the range indexed strings which belong to K. This function belongs to `bad_base_field.c`. It is run by 66 tests, including 61 in the `bad` library and 5 in the `DifferentialAlgebra` package.

`void bad_set_base_field_relations_properties (struct bad_regchain *relations, bool differential)` [Function]

If *relations* is empty, set all its properties (except the differential and the coherence ones if *differential* is false). If it is nonempty, set the prime ideal property then call `bad_set_automatic_properties_attchain`. The parameter *relations* is supposed to provide the `relations` field of a base field. This function belongs to `bad_base_field.c`. It is run by 229 tests, including 114 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`void bad_set_base_field_generators_and_relations (struct bad_base_field *K, struct ba0_tableof_range_indexed_group *generators, struct bad_regchain *relations, bool pretend)` [Function]

Assign to K the base field defined by *generators* and *relations*.

The *generators* table is supposed to contain a subset of the entries of the block tables of orderings. All the symbols which occur in *generators* and *relations* are moved to the base field.

If *pretend* is `false` then some checking is performed to ensure the consistency of the relations. The arguments *generators* and *relations* may be zero. This function belongs to `bad_base_field.c`. It is run by 84 tests, including 62 in the `bad` library and 22 in the `DifferentialAlgebra` package.

`bool bad_is_a_compatible_base_field (struct bad_base_field *K, struct bad_attchain *attrib)` [Function]

Return `true` if the relations of K can be included in a regular chain having properties *attrib*. See `bad_is_a_compatible_regchain`. This function belongs to `bad_base_field.c`. It is run by 147 tests, including 68 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`bool bad_member_variable_base_field (struct bav_variable *v, struct bad_base_field *K)` [Function]

Return `true` if v belongs to K, else `false`. This function belongs to `bad_base_field.c`. It is run by 188 tests, including 94 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`ba0_int_p bad_number_of_elements_over_base_field_regchain (struct bad_regchain *A, struct bad_base_field *K)` [Function]

Return the number of elements of A which do not belong to K. This function belongs to `bad_base_field.c`. It is run by 6 tests, including 4 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`bool bad_member_nonzero_polynom_base_field (struct bap_polynom_mpz *P, struct bad_base_field *K)` [Function]

Return `true` if P is a nonzero element of K. If the field `assume_reduced` of K is `true`, then the polynomial P is supposed to be reduced with respect to the defining relations of K, thereby speeding up the membership test. This function belongs to `bad_base_field.c`. It is run by 152 tests, including 73 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`bool bad_member_polynom_base_field (struct bap_polynom_mpz *P, struct bad_base_field *K)` [Function]

Return `true` if P is an element (possibly zero) of K. This function belongs to `bad_base_field.c`. It is run by 188 tests, including 95 in the `bad` library and 31 in the `DifferentialAlgebra` package.

`bool bad_member_product_base_field (struct bap_product_mpz *P, struct bad_base_field *K)` [Function]

Return `true` if P belongs to K, else `false`. This function belongs to `bad_base_field.c`. It is run by 53 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_remove_product_factors_base_field (struct bap_product_mpz *R, struct bap_product_mpz *P, struct bad_base_field *K)` [Function]

Remove from the product P all factors which belong to K and set to 1 the exponents of the remaining factors. Set to 1 the numerical factor. Assign the result to R. This function belongs to `bad_base_field.c`. It is run by 53 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void * bad_scanf_base_field (void *AA)` [Function]

The general parsing function for base fields. It is called by `ba0_scanf/%base_field`. The input string is expected to have the form: `field ( [ differential = boolean ], [ generators = %t[%range_indexed_group] ], [ relations = %regchain ] )`. The entries of the table of generators should be a subset of the ones of the tables of blocks of orderings. The table does not need to be exhaustive since any variable which belongs to a block lower than the block of any other base field element is automatically considered as a base field element. The flag `differential` permits to force a base field to be non differential even in the differential context i.e. when at least one derivation is defined.

Keywords `basefield` and `base_field` may be used instead of `field`. Exception `BAD_ERRBAS` may be raised. This function belongs to `bad_base_field.c`. It is run by 54 tests, including 54 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_printf_base_field (void *AA)` [Function]

A printing function for base fields. It is called by `ba0_printf/%base_field`. This function belongs to `bad_base_field.c`. It is run by 61 tests, including 61 in the `bad` library and 0 in the `DifferentialAlgebra` package.

8.8 Reduction

Description of type `bad_typeof_reduction_strategy`. This data type permits to choose the method carried out by the reduction algorithm. This type declaration belongs to `bad_reduction.h`.

```
enum bad_typeof_reduction_strategy
{
// the polynomial to be reduced is factored/easy then reduction is performed factorwise and
// gcd are computed to reduce power products of initial and separants involved in the process
// - default value
  bad_gcd_prem_and_factor_reduction_strategy,
// basic strategy
  bad_prem_reduction_strategy,
// a change of ordering is performed first in order
// to reduce polynomials coefficient per coefficient
  bad_prem_and_change_of_ordering_reduction_strategy,
};
```

Description of type `bad_typeof_redzero_strategy`. This data type permits to choose the method carried out by the algorithms designed for testing if a differential polynomial gets reduced to zero or not. This type declaration belongs to `bad_reduction.h`.

```
enum bad_typeof_redzero_strategy
{
// the result is guaranteed - default value
  bad_deterministic_using_probabilistic_redzero_strategy,
// perform reduction and test if the result is zero
  bad_deterministic_redzero_strategy,
// the result is not guaranteed
  bad_probabilistic_redzero_strategy
};
```

`void bad_set_settings_reduction (enum bad_typeof_reduction_strategy` *`reduction_strategy`*`, enum bad_typeof_redzero_strategy` *`redzero_strategy`*`, ba0_int_p` *`number_of_redzero_tries`*`)` [Function]

Set `bad_reduction_strategy`, `bad_redzero_strategy` and `bad_number_of_redzero_tries` to *reduction_strategy*, *redzero_strategy* and *number_of_redzero_tries*. Zero values are replaced by default values. This function belongs to `bad_reduction.c`. It is run by 94 tests, including 29 in the `bad` library and 65 in the `DifferentialAlgebra` package.

`void bad_get_settings_reduction (enum bad_typeof_reduction_strategy *`*`reduction_strategy`*`, enum bad_typeof_redzero_strategy *`*`redzero_strategy`*`, ba0_int_p *`*`number_of_redzero_tries`*`)` [Function]

Assign to *reduction_strategy*, *redzero_strategy* and *number_of_redzero_tries* the values of the corresponding setting variables. This function belongs to `bad_reduction.c`. It is not run by any test.

Description of type `bad_typeof_reduction`. This data type permits to specify to reduction algorithms the type of reduction to be performed. This type declaration belongs to `bad_reduction.h`.

```
enum bad_typeof_reduction
{
  bad_full_reduction,
  bad_partial_reduction,
// no derivation is performed
  bad_algebraic_reduction
};
```

Description of type `bad_typeof_derivative_to_reduce`. This data type permits to indicate to reduction algorithms if all derivatives have to be reduced or if the leading derivative has to be preserved. This type declaration belongs to `bad_reduction.h`.

```
enum bad_typeof_derivative_to_reduce
{
  bad_all_derivatives_to_reduce,
  bad_all_but_leader_to_reduce
};
```

`bool bad_is_a_reducible_polynom_by_regchain (struct bap_polynom_mpz *A, struct bad_regchain *C, enum bad_typeof_reduction type_red, enum bad_typeof_derivative_to_reduce type_der, struct bav_rank *rg, ba0_int_p *index)` [Function]

Return `true` if C permits to reduce A with respect to the reduction defined by *type_red* and *type_der*. If so *rg* receives the highest reducible rank and *index* receives the index of some element of C able to reduce this rank. The two last parameters may be the zero pointer. This function belongs to `bad_reduction.c`. It is run by 235 tests, including 118 in the `bad` library and 34 in the `DifferentialAlgebra` package.

`bool bad_is_a_reducible_product_by_regchain (struct bap_product_mpz *A, struct bad_regchain *C, enum bad_typeof_reduction type_red, enum bad_typeof_derivative_to_reduce type_der, ba0_int_p *index)` [Function]

Variant of the `bad_is_a_reducible_polynom_by_regchain`, for products. This function belongs to `bad_reduction.c`. It is run by 208 tests, including 112 in the `bad` library and 32 in the `DifferentialAlgebra` package.

`bool bad_is_a_partially_reduced_polynom_wrt_regchain (struct bap_polynom_mpz *A, struct bad_regchain *C)` [Function]

Return `true` if A is partially reduced with respect to C i.e. if A does note involve any proper derivative of any leader of C. This function belongs to `bad_reduction.c`. It is run by 168 tests, including 67 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_reduce_easy_polynom_by_regchain (struct bap_polynom_mpz *R, struct bap_polynom_mpz *A, struct bad_regchain *C, enum bad_typeof_reduction type_red)` [Function]

Reduce A by the elements of C which are likely to decrease the number of monomials of A. The reduction is purely algebraic. The result is stored in R and H. This function belongs to `bad_reduction.c`. It is run by 157 tests, including 85 in the `bad` library and 27 in the `DifferentialAlgebra` package.

`void bad_ensure_nonzero_initial_mod_regchain (struct bap_polynom_mpz *R, struct bap_polynom_mpz *A0, struct bad_regchain *C, enum bad_typeof_reduction type_red)` [Function]

Replace *A0* by its reductum while its initial is reduced to zero by C, using the type of reduction *type_red*. Result in R. This function belongs to `bad_reduction.c`. It is run by 147 tests, including 77 in the `bad` library and 27 in the `DifferentialAlgebra` package.

`void bad_reduce_polynom_by_regchain (struct bap_product_mpz *R, struct bap_product_mpz *H, struct bav_tableof_term *theta, struct bap_polynom_mpz *A, struct bad_regchain *C, enum bad_typeof_reduction type_red, enum bad_typeof_derivative_to_reduce type_der)` [Function]

Reduce A by C. Result in R, H and *theta*.

If *type_red* is `bad_algebraic_reduction` then the function computes a relation $H\,A = R$ modulo the algebraic ideal (C) where H is a power product of initials of elements of C and R is algebraically reduced with respect to C.

Otherwise, the function computes a relation $H\,A = R$ modulo the differential ideal $[C]$.

If *type_red* is `bad_partial_reduction` then H is a power product of separants of elements of C and R is partially reduced with respect to C.

If *type_red* is `bad_full_reduction` then H is a power product of initials and separants of C and R is fully reduced with respect to C.

The table *theta* receives a table of terms of the same size as the decision system of the regular chain C. The ith term of *theta* corresponds to the ith element of C and receives the least common multiple of all the derivation operators which were applied to the element of C during the reduction process. Of course, if *type_red* is `bad_algebraic_reduction` then all the terms of *theta* are set to 1.

The parameter *type_der* indicates if the leader of A is concerned by the reduction or not.

Parameters R, H and *theta* may be zero pointers. The case where the ordering of C is different from the current ordering is handled. This function belongs to `bad_reduction.c`. It is run by 214 tests, including 117 in the `bad` library and 33 in the `DifferentialAlgebra` package.

`void bad_reduce_product_by_regchain (struct bap_product_mpz *R, struct bap_product_mpz *H, struct bav_tableof_term *T, struct bap_product_mpz *A, struct bad_regchain *C, enum bad_typeof_reduction type_red, enum bad_typeof_derivative_to_reduce type_der)` [Function]

Variant of `bad_reduce_polynom_by_regchain`, for products. This function belongs to `bad_reduction.c`. It is run by 55 tests, including 8 in the `bad` library and 14 in the `DifferentialAlgebra` package.

`bool bad_is_a_reduced_to_zero_polynom_by_regchain (struct bap_polynom_mpz *A, struct bad_regchain *C, enum bad_typeof_reduction type_red)` [Function]

Return `true` if A is probably reduced to zero by C by means of a reduction of type *type_red*. The case where the ordering of C is different from the current ordering is handled. Exception `BAO_ERRALG` is raised if *type_red* is `bad_partial_reduction`. This function belongs to `bad_reduction.c`. It is run by 183 tests, including 92 in the `bad` library and 31 in the `DifferentialAlgebra` package.

8.9 Regularization

Description of type `bad_typeof_regularize_strategy`. This data type permits to indicate to regularization algorithms the strategy to be carried out. All tests are based on pseudo-remainder sequence computations. This type declaration belongs to `bad_regularize.h`.

```
enum bad_typeof_regularize_strategy
{
// Lionel Ducos algorithm is applied
```

```
  bad_subresultant_regularize_strategy = 1,
// pseudoremainders are computed using baz_gcd_pseudo_division_polynom_mpz
  bad_gcd_prem_regularize_strategy,
// pseudoremainders are computed factor per factor using
//  baz_gcd_pseudo_division_polynom_mpz
  bad_gcd_prem_and_factor_regularize_strategy
};
```

`void bad_set_settings_regularize (enum bad_typeof_regularize_strategy strategy)` [Function]

Set `bad_regularize_strategy` to *strategy*. A zero value is replaced by the default value. This function belongs to `bad_regularize.c`. It is run by 65 tests, including 0 in the `bad` library and 65 in the `DifferentialAlgebra` package.

`void bad_get_settings_regularize (enum bad_typeof_regularize_strategy *strategy)` [Function]

Assign `bad_regularize_strategy` to **strategy*. The pointer may be zero. This function belongs to `bad_regularize.c`. It is not run by any test.

Description of type `bad_typeof_Euclid`. This data type permits to control the type of extended Euclidean algorithm to be performed. This type declaration belongs to `bad_regularize.h`.

```
enum bad_typeof_Euclid
{
  bad_basic_Euclid,
  bad_half_extended_Euclid,
  bad_extended_Euclid
};
```

`void bad_Euclid_mod_regchain (struct bap_tableof_tableof_polynom_mpz *tabV, struct bad_tableof_quadruple *tabG, enum bad_typeof_Euclid type, struct bap_polynom_mpz *A, struct bap_polynom_mpz *B, struct bav_variable *v, bool reginitA, bool reginitB, struct bad_regchain *C, struct bad_base_field *K, struct bap_polynom_mpz * *ddz)` [Function]

This function essentially performs the Euclidean algorithm (possibly extended or half-extended) over A and B, with coefficients taken modulo some ideal I, which is precised below.

It used to be called from different contexts but it is now only called from `bad_pardi`.

The polynomials are supposed to have coefficients in the field K.

The variable v is assumed to be greater than or equal to the leaders of these two polynomials. The booleans *reginitA* and *reginitB* indicate if the leading coefficients of A and B are regular modulo I. In some cases, exhibiting a zero divisor modulo I raises the exception `BAD_EXRDDZ`. The zero divisor is then stored in *ddz*.

The last entry of *tabV* receives an Euclidean relation. If *type* is `bad_extended_Euclid` then triples (G, U, V) such that $U\,A + V\,B = G$ are stored in *tabV*. If *type* is `bad_basic_Euclid` then only G is computed. If *type* is `bad_half_extended_Euclid` then only G and U are computed.

The parameter *tabV* is a table because the computation may lead to splitting cases (in which case, new entries need to be appended) or to some inconsistencies (in which case, entries need to be removed). Contexts are as follows.

- When called from `bad_Rosenfeld_Groebner`, the parameter C is zero and the table *tabG* is not zero. Then the ideal I is the ideal associated to the last entry G of *tabG*. Booleans *reginitA* and *reginitB* just indicate if the leading coefficients of A and B are stored in the inequations list of G. Each step of the Euclidean algorithm normally leads to a splitting: the leading coefficient of the current remainder is considered as nonzero in the general branch; unless discarded by the `bad_nonzero` function, it is also separately considered as zero, producing another quadruple in *tabG* and another entry in *tabV*. All branches are processed up to the end. Some of them (possibly all) can be proven inconsistent. The corresponding entries of *tabG* and *tabV* are then removed.
- When called from `bad_pardi`, the parameters C and *tabG* are nonzero. The ideal I is the one defined by C. It is assumed to be prime. The `bad_pardi` function only needs one quadruple, which is the last entry of *tabG*.
- When called from `bad_invert_polynom_mod_regchain`, the parameter C is nonzero while *tabG* is zero. The ideal I is the one defined by C. It does not need to be prime. The regularity tests may exhibit some zero divisor. In such cases, the exception `BAD_EXRDDZ` is raised and the zero divisor is stored in **ddz*.

This function belongs to `bad_regularize.c`. It is run by 2 tests, including 1 in the `bad` library and 1 in the `DifferentialAlgebra` package.

`void bad_check_regularity_polynom_mod_regchain (struct bap_polynom_mpz *A, struct bad_regchain *C, struct bad_base_field *K, struct bap_polynom_mpz * *ddz)` [Function]

Return if A is regular modulo the ideal defined by C over the base field K.

Exception `BAD_EXRNUL` is raised if A is zero modulo C.

Exception `BAD_EXRDDZ` is raised if a zerodivisor modulo C is exhibited during the computation. This zerodivisor may be different from A. In this case and if *ddz* is nonzero, then the zerodivisor is returned in *ddz* and is guaranteed to have a regular initial modulo C. This function belongs to `bad_regularize.c`. It is run by 187 tests, including 97 in the `bad` library and 31 in the `DifferentialAlgebra` package.

`void bad_resultant_mod_regchain (struct bap_product_mpz *R, struct bap_polynom_mpz *P, struct bad_regchain *A)` [Function]

Assign to R the resultant of P with respect to the regular chain A. If A is differential then P is replaced by its partial remainder with respect to A before the resultant computation. The resulting polynomial is zero if and only if P is a zero divisor modulo the ideal defined by A. This function belongs to `bad_resultant.c`. It is run by 7 tests, including 6 in the `bad` library and 1 in the `DifferentialAlgebra` package.

`void bad_reg_characteristic_regchain (struct` [Function]
`bad_intersectof_regchain *tabC, struct bap_listof_polynom_mpz *S0, struct bad_regchain *ideal, struct bad_base_field *K)`

Denote J the unit ideal if *ideal* is zero and the ideal defined by *ideal* otherwise. Denote C the last element of *tabC*.

Let us first consider the differential context. It is assumed that $[C] : H_C^\infty \subset [C] : S^\infty$ and that the elements of S are partially reduced with respect to C. This function stores in *tabC*, at indices greater than or equal to that of C, regular chains $A_1, \ldots, A_t$ such that $[C] : H_C^\infty \subset [C] : S^\infty \subset [A_1] : H_{A_1}^\infty \cap \cdots \cap [A_t] : H_{A_t}^\infty \subset J$.

In the algebraic context, the function stores in *tabC*, at indices greater than or equal to that of C, regular chains $A_1, \ldots, A_t$ such that $(C) : I_C^\infty \subset (C) : S^\infty \subset (A_1) : I_{A_1}^\infty \cap \cdots \cap (A_t) : I_{A_t}^\infty \subset J$.

In the case J is a prime ideal then only one regular chain is computed.

If moreover J is minimal over the ideal defined by C then the resulting regular chain is a characteristic set of J.

Polynomials are supposed to have coefficients in K. This function belongs to `bad_regularize.c`. It is run by 149 tests, including 71 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_invert_polynom_mod_regchain (struct` [Function]
`bap_product_mpz *U, struct bap_product_mpz *G, struct bap_polynom_mpz *A, struct bad_regchain *C, struct bad_base_field *K, struct bap_polynom_mpz *volatile *ddz)`

Compute U and G such that a relation $U\,A = G$ holds modulo the ideal defined by C, over the base field K. The fraction U/G may be viewed as an inverse of A modulo C. In particular, G is regular modulo C.

Exception `BAD_EXRNUL` is raised if A is zero modulo C.

Exception `BAD_EXRDDZ` is raised if a zerodivisor modulo C is exhibited during the computation. This zerodivisor may be different from A. In this case and if *ddz* is nonzero, then the zerodivisor is returned in *ddz* and is guaranteed to have a regular initial modulo C. This function belongs to `bad_invert.c`. It is run by 98 tests, including 29 in the `bad` library and 24 in the `DifferentialAlgebra` package.

`void bad_invert_product_mod_regchain (struct` [Function]
`bap_product_mpz *U, struct bap_product_mpz *G, struct bap_product_mpz *A, struct bad_regchain *C, struct bad_base_field *K, struct bap_polynom_mpz *volatile *ddz)`

Variant of `bad_invert_polynom_mod_regchain` for products. The inverses are computed factorwise. This function belongs to `bad_invert.c`. It is run by 55 tests, including 8 in the `bad` library and 14 in the `DifferentialAlgebra` package.

8.10 Normal forms

The normal form of a polynomial (or more generally, a rational fraction) A modulo the ideal defined by a regular chain C is a rational fraction. In the differential case, the rational

fraction is partially reduced w.r.t. C. Its denominator does not divide zero modulo the ideal defined by C and does not depend on the leading derivatives of the elements of C. Mathematically, the normal form is a canonical representative of the residue class of A in the quotient ring defined by C.

In general, normal form computations may fail. If the failure is due to the fact that the denominator of A is zero modulo the ideal defined by C, then the normal form is undefined. Otherwise, the failure may be due to an exhibited factorization of some element of C. In this case, the regular chain C can be split in many different chains. The normal form of A may or may not exist w.r.t. to each of these chains.

In the simpler case of A being a polynomial, the normal form can be computed with no failure.

Note: when a chain C is split as two chains C_1 and C_2 then the radical of the ideal defined by C is the intersection of the radicals of the ideals defined by C_1 and C_2. If the chain C is squarefree (in particular, when it is differential) then the ideal defined by C is the intersection of the ideals defined by C_1 and C_2.

`void bad_normal_form_polynom_mod_regchain (struct baz_ratfrac *NF, struct bap_polynom_mpz *A, struct bad_regchain *C, struct bap_polynom_mpz * *ddz)` [Function]

Assign to *NF* the normal form of A modulo the ideal I defined by C or raise an exception. The function calls `bad_invert_polynom_mod_regchain`. It may thus fail and raise the exceptions `BAD_EXRNUL` or `BAD_EXRDDZ`. This function belongs to `bad_regularize.c`. It is run by 38 tests, including 3 in the `bad` library and 9 in the `DifferentialAlgebra` package.

`void bad_normal_form_ratfrac_mod_regchain (struct baz_ratfrac *NF, struct baz_ratfrac *A, struct bad_regchain *C, struct bap_polynom_mpz * *ddz)` [Function]

Variant of `bad_reduced_form_polynom_mod_regchain` for rational fractions. This function belongs to `bad_regularize.c`. It is run by 69 tests, including 10 in the `bad` library and 14 in the `DifferentialAlgebra` package.

`void bad_normal_form_ratfrac_mod_intersectof_regchain (struct baz_tableof_ratfrac *tabNF, struct baz_ratfrac *A, struct bad_intersectof_regchain *tabC, struct bap_polynom_mpz * *ddz)` [Function]

Compute one normal form of A for each regchain in *tabC* Append the normal forms to *tabNF*. This function belongs to `bad_regularize.c`. It is run by 4 tests, including 0 in the `bad` library and 4 in the `DifferentialAlgebra` package.

`void bad_normal_form_handling_exceptions_ratfrac_mod_regchain (struct baz_tableof_ratfrac *tabNF, struct bad_intersectof_regchain *tabC, struct bad_intersectof_regchain *tabNUL, struct baz_ratfrac *A)` [Function]

Denote C the regular chain on the top of *tabC*. Compute the normal form of A modulo C. In most cases, the computation is successful and the resulting normal form is just stored on the top of *tabNF*.

In some cases however, the computation may raise one of the exceptions `BAD_EXRNUL` or `BAD_EXRDDZ`, leading to splitting cases. The regular chains produced by the splittings are stored either on the top of *tabC* (starting from the index of *C*, since this chain is removed in the case of splittings) or on the top of *tabNUL*. The chains which are stored in *tabC* are those with respect to which normal forms can eventually be computed. The chains which are stored in *tabNUL* are those with respect to which the denominator of *A* is zero.

At the end of the process, the radical of the ideal defined by *C* is equal to the radical of the ideals defined by the chains stored in *tabC* and *tabNUL*.

For each regular chain stored in *tabC*, the normal form of *A* is stored in *tabNF*.

Exceptions are handled as follows. If `BAD_EXRNUL` is raised then the denominator of *A* is zero modulo the current chain. This chain is stored in *tabNUL*. If `BAD_EXRDDZ` is raised then `bad_handle_splitting_exceptions_regchain` is called in order to split the current chain and the normal form computations are restarted modulo the new regular chains. The corresponding normal forms of *A* are stored on the top of *tabNF*. This function belongs to `bad_regularize.c`. It is run by 12 tests, including 12 in the `bad` library and 0 in the `DifferentialAlgebra` package.

8.11 Reduced forms

`void bad_reduced_form_polynom_mod_regchain (struct baz_ratfrac *R, struct bap_polynom_mpz *F, struct bav_variable *v, struct bad_regchain *C)` [Function]

Assign to *R* a reduced form of *F* with respect to *C*. If *v* is zero, the reduced form is a polynomial. If *v* is nonzero, the reduced form is allowed to have a denominator of rank strictly less than *v*.

More generally, the reduced form $RF(F)$ of F is defined as follows: if the denominator of $NF(F)$ has rank less than v then $RF(F) = NF(F)$ otherwise, decomposing $F = i_F\, u^d + G$, we have $RF(F) = RF(i_F)\, RF(u^d) + RF(G)$. This function belongs to `bad_reduced_form.c`. It is run by 1 tests, including 1 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_reduced_form_polynom_mod_intersectof_regchain (struct baz_tableof_ratfrac *tabR, struct bap_polynom_mpz *F, struct bav_variable *v, struct bad_intersectof_regchain *tabC)` [Function]

Assign to *tabR* the reduced forms of *F* w.r.t. each component of *tabC*. This function belongs to `bad_reduced_form.c`. It is not run by any test.

8.12 Critical pairs

Description of type `bad_typeof_critical_pair`. This data type permits to tag critical pairs in order to process the most important ones before the other ones. The default tag is `bad_normal_critical_pair`. The tag `bad_rejected_easy_critical_pair` corresponds to critical pairs $\{p_1, p_2\}$ which are not reduction critical pairs and such that neither *p_1* nor *p_2* occurs in the field `A` of the current quadruple. This type declaration belongs to `bad_critical_pair.h`.

```
enum bad_typeof_critical_pair
{
// default tag
  bad_normal_critical_pair,
// this tag indicates a lower priority
  bad_rejected_easy_critical_pair
};
```

Description of type `bad_critical_pair`. This data type implements critical pairs. A pair $\{p_1, p_2\}$ of differential polynomials is said to be a *critical pair* if the leaders of p_1 and p_2 are derivatives of some same differential indeterminate u. Denote $\theta_1 u$ the leading derivative of p_1 and $\theta_2 u$ the one of p_2.

Denote $\theta_{12} = lcm(\theta_1, \theta_2)$.

If $\theta_{12} = \theta_1$ or $\theta_{12} = \theta_2$ then the pair is called a *reduction critical pair* and the corresponding Δ-polynomial is $\Delta(p_1, p_2) = prem(p_2, (\theta_{12}/\theta_1)p_1)$

If the critical pair is not a reduction one then the corresponding Δ-polynomial is $\Delta(p_1, p_2) = s_2\,(\theta_{12}/\theta_1)p_1 - s_1\,(\theta_{12}/\theta_2)p_2$ where s_1 and s_2 denote the separants of p_1 and p_2. This type declaration belongs to `bad_critical_pair.h`.

```
struct bad_critical_pair
{
  enum bad_typeof_critical_pair tag;
  struct bap_polynom_mpz p;
  struct bap_polynom_mpz q;
};
```

`void bad_init_critical_pair (struct bad_critical_pair *pair)` [Function]

Initialize *pair* to an empty (meaningless) critical pair. This function belongs to `bad_critical_pair.c`. It is run by 132 tests, including 60 in the `bad` library and 27 in the `DifferentialAlgebra` package.

`struct bad_critical_pair * bad_new_critical_pair (void)` [Function]

Allocate a new critical pair, initialize it and return it. This function belongs to `bad_critical_pair.c`. It is run by 132 tests, including 60 in the `bad` library and 27 in the `DifferentialAlgebra` package.

`struct bad_critical_pair * bad_new_critical_pair_polynom_mpz (struct bap_polynom_mpz *p, struct bap_polynom_mpz *q)` [Function]

Return the new critical pair $\{p, q\}$. Exception `BAD_ERRCRI` is raised if $\{p, q\}$ is not a critical pair. This function belongs to `bad_critical_pair.c`. It is run by 132 tests, including 60 in the `bad` library and 27 in the `DifferentialAlgebra` package.

`void bad_set_critical_pair (struct bad_critical_pair *dst, struct bad_critical_pair *src)` [Function]

Assign *src* to *dst*. This function belongs to `bad_critical_pair.c`. It is run by 132 tests, including 60 in the `bad` library and 27 in the `DifferentialAlgebra` package.

`void bad_set_critical_pair_polynom_mpz (struct bad_critical_pair *pair, struct bap_polynom_mpz *p, struct bap_polynom_mpz *q)` [Function]

Assign to *pair* the critical pair $\{p, q\}$. Exception `BAD_ERRCRI` is raised if $\{p, q\}$ is not a critical pair. This function belongs to `bad_critical_pair.c`. It is run by 132 tests, including 60 in the `bad` library and 27 in the `DifferentialAlgebra` package.

`bool bad_is_an_algebraic_critical_pair (struct bad_critical_pair *pair)` [Function]

Return `true` if the two members of *pair* have the same leader. This function belongs to `bad_critical_pair.c`. It is run by 121 tests, including 51 in the `bad` library and 25 in the `DifferentialAlgebra` package.

`bool bad_is_a_reduction_critical_pair (struct bad_critical_pair *pair, struct bav_variable **var)` [Function]

Return `true` if *pair* is a reduction critical pair. If it is and *var* is nonzero, assign to **var* the highest of the leaders of the two pair elements. This function belongs to `bad_critical_pair.c`. It is run by 28 tests, including 14 in the `bad` library and 14 in the `DifferentialAlgebra` package.

`void bad_delta_polynom_critical_pair (struct bap_polynom_mpz *delta, struct bad_critical_pair *pair)` [Function]

Assign to *delta* the Δ-polynomial generated by the critical pair. This function belongs to `bad_critical_pair.c`. It is run by 130 tests, including 59 in the `bad` library and 26 in the `DifferentialAlgebra` package.

`void bad_thetas_and_leaders_critical_pair (struct bav_tableof_term *thetas, struct bav_tableof_variable *leaders, struct bad_critical_pair *pair)` [Function]

Assign to *leaders* the leading derivatives of the elements of *pair* that need be differentiated in order to compute the Δ-polynomial and assign to *thetas* the corresponding derivation operators. This function belongs to `bad_critical_pair.c`. It is run by 49 tests, including 3 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`bool bad_is_a_simpler_critical_pair (struct bad_critical_pair *P, struct bad_critical_pair *Q, struct bad_selection_strategy *strategy)` [Function]

Heuristic function which returns `true` if P should be considered before Q in a list of critical pairs waiting for being processed, according to *strategy*. This function is used to sort the lists of critical pairs occurring in quadruples. It is based on estimates of the sizes of the Δ-polynomials. Critical pairs whose Δ-polynomials have fewer monomials are considered simpler. This function belongs to `bad_critical_pair.c`. It is run by 88 tests, including 32 in the `bad` library and 24 in the `DifferentialAlgebra` package.

`bool bad_is_a_listof_rejected_critical_pair (struct bad_listof_critical_pair *L)` [Function]

Return `true` if the elements of L are all rejected critical pairs. This function is used by `bad_pardi` and `bad_Rosenfeld_Groebner` to stop their main loops. This function belongs to `bad_critical_pair.c`. It is run by 154 tests, including 75 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void * bad_scanf_critical_pair (void *A)` [Function]
The parsing function for critical pairs. It is called by `ba0_scanf/%critical_pair`. This function belongs to `bad_critical_pair.c`. It is not run by any test.

`void bad_printf_critical_pair (void *A)` [Function]
The printing function for critical pairs. It is called by `ba0_printf/%critical_pair`. This function belongs to `bad_critical_pair.c`. It is not run by any test.

8.13 Selection strategies

Description of type `bad_typeof_selection_strategy`. This data type is used to specify a strategy, carried out by differential elimination algorithm, in order to select the next polynomial to process. This type declaration belongs to `bad_selection_strategy.h`.

```
enum bad_typeof_selection_strategy
{
// plain polynomials are preferred to polynomials arising from critical pairs
  bad_equation_first_selection_strategy,
// polynomials with lower leaders are preferred
  bad_lower_leader_first_selection_strategy
};
```

Description of type `bad_selection_strategy`. This data type specifies a strategy, carried out by differential elimination algorithm, in order to select the next polynomial to process. This type declaration belongs to `bad_selection_strategy.h`.

```
struct bad_selection_strategy
{
  enum bad_typeof_selection_strategy strategy;
// a penalty used to penalize some critical pairs
  ba0_int_p penalty;
};
```

`void bad_init_selection_strategy (struct bad_selection_strategy *S)` [Function]
Initialize S with `strategy` equal to `bad_lower_leader_first_selection_strategy` and `penalty` equal to 2. This function belongs to `bad_selection_strategy.c`. It is run by 156 tests, including 77 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`struct bad_selection_strategy * bad_new_selection_strategy (void)` [Function]
Allocate a new `struct bad_selection_strategy`, initialize it and return it. This function belongs to `bad_selection_strategy.c`. It is run by 156 tests, including 77 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_strategy_selection_strategy (struct bad_selection_strategy *S, enum bad_typeof_selection_strategy strategy)` [Function]
Set the field `strategy` of S to *strategy*. This function belongs to `bad_selection_strategy.c`. It is run by 11 tests, including 9 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_set_penalty_selection_strategy (struct bad_selection_strategy *S, ba0_int_p penalty)` [Function]

Set the field `penalty` of S to *penalty*. This function belongs to `bad_selection_strategy.c`. It is not run by any test.

`void bad_double_penalty_selection_strategy (struct bad_selection_strategy *S)` [Function]

Double the value of the field `penalty` of S, unless an overflow occurs. This function belongs to `bad_selection_strategy.c`. It is run by 3 tests, including 2 in the `bad` library and 1 in the `DifferentialAlgebra` package.

8.14 Splitting trees

Description of type `bad_activity_level_splitting_tree`. This data type is a subtype of `bad_splitting_tree`. This type declaration belongs to `bad_splitting_tree.h`.

```
enum bad_activity_level_splitting_tree
{
// the splitting tree is not generated
  bad_inactive_splitting_tree,
// the splitting tree is generated
  bad_quiet_splitting_tree,
// the splitting tree is generated and messages are printed
  bad_verbose_splitting_tree
};
```

Description of type `bad_splitting_tree`. This data type permits to describe the splitting tree generated by a differential elimination algorithm.

The field `vertices` contains the table of the tree vertices. Each vertex is associated to one quadruple / regular chain, identified by its *number*. This *number* is moreover equal to the index of the vertex in the table.

The first entry of `vertices`, with *number* zero, is the *root vertex*. It does not correspond to any actual quadruple / regular chain. It permits to handle systems which have many different *first* vertices. First vertices play a special role in differential elimination methods because they provide bounds which permit to discard quadruples by means of a dimension argument.

The field `number` contains the next free vertex number. It may be greater than the `alloc` field of the `vertices` table.

The field `activity` provides the level of activity of the tree. It may have three values:

- `bad_inactive_splitting_tree` then at most one vertex is allocated to `vertices`: the *root* vertex
- `bad_quiet_splitting_tree` then the splitting tree is generated
- `bad_verbose_splitting_tree` same as above but the differential elimination algorithm which builds the tree may print some data on the standard output.

This type declaration belongs to `bad_splitting_tree.h`.

```
struct bad_splitting_tree
{
  enum bad_activity_level_splitting_tree activity;
// the table of vertices
  struct bad_tableof_splitting_vertex vertices;
```

```
// the next free vertex number
  ba0_int_p number;
};
```

`void bad_init_splitting_tree (struct bad_splitting_tree *tree)` [Function]

Initialize *tree* to an empty inactive splitting tree. This function belongs to `bad_splitting_tree.c`. It is run by 156 tests, including 77 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`struct bad_splitting_tree * bad_new_splitting_tree (void)` [Function]

Allocate a new splitting tree, initialize it and return it. This function belongs to `bad_splitting_tree.c`. It is run by 105 tests, including 26 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_reset_splitting_tree (struct bad_splitting_tree *tree, enum bad_activity_level_splitting_tree level)` [Function]

Reset *tree* to an empty splitting tree with activity level *level*. This function belongs to `bad_splitting_tree.c`. It is run by 116 tests, including 55 in the `bad` library and 12 in the `DifferentialAlgebra` package.

`void bad_set_splitting_tree (struct bad_splitting_tree *dst, struct bad_splitting_tree *src)` [Function]

Assign *src* to *dst*. This function belongs to `bad_splitting_tree.c`. It is not run by any test.

`ba0_int_p bad_next_number_splitting_tree (struct bad_splitting_tree *tree)` [Function]

Return the next available quadruple or regular chain number available in *tree*. This function does not allocate any memory. This function belongs to `bad_splitting_tree.c`. It is run by 154 tests, including 75 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`struct bad_splitting_vertex * bad_ith_vertex_splitting_tree (struct bad_splitting_tree *tree, ba0_int_p number)` [Function]

Return the splitting vertex dedicated to the quadruple or the regular chain identified by *number* in *tree*. This function may perform memory allocation. This function belongs to `bad_splitting_tree.c`. It is run by 145 tests, including 66 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`bool bad_is_first_vertex_splitting_tree (struct bad_splitting_tree *tree, ba0_int_p number)` [Function]

Return `true` if *number* is the number of a *first* vertex else `false`. This function belongs to `bad_splitting_tree.c`. It is not run by any test.

`void bad_set_first_vertex_splitting_tree (struct bad_splitting_tree *tree, ba0_int_p number, bool is_first)` [Function]

Set the `is_first` field of the vertex of *tree* with number *number* to *is_first*. If *is_first* is `true` then the vertex becomes a *first* vertex. This function belongs to `bad_`

splitting_tree.c. It is run by 145 tests, including 66 in the bad library and 30 in the DifferentialAlgebra package.

void bad_add_edge_splitting_tree (struct bad_splitting_tree *tree, enum bad_typeof_splitting_edge type, ba0_int_p src, ba0_int_p dst, struct bav_variable *leader, ba0_int_p multiplicity) [Function]

Add the edge defined by the five last arguments to *tree*. In the case of an inactive splitting tree, the edge is not recorded. If an edge from *src* to *dst* has already been defined then it must have the same type as *type* and the same leader field as *leader*. Exception BA0_ERRALG is raised if *src* and *dst* are equal. This function belongs to bad_splitting_tree.c. It is run by 145 tests, including 66 in the bad library and 30 in the DifferentialAlgebra package.

void bad_add_edge_novar_splitting_tree (struct bad_splitting_tree *tree, enum bad_typeof_splitting_edge type, ba0_int_p src, ba0_int_p dst) [Function]

Add the edge defined by *type*, *src* and *dst* to *tree*. In the case of an inactive splitting tree, the edge is not recorded. Exception BA0_ERRALG is raised if *src* and *dst* are equal. This function belongs to bad_splitting_tree.c. It is run by 145 tests, including 66 in the bad library and 30 in the DifferentialAlgebra package.

void bad_merge_thetas_leaders_vertex_splitting_tree (struct bad_splitting_tree *tree, ba0_int_p number, struct bav_tableof_term *thetas, struct bav_tableof_variable *leaders) [Function]

Let V denote the vertex of *tree* with number *number*. Merge the field leaders of V with *leaders* so that the resulting field leaders of V remains sorted by decreasing order. Perform corresponding operations on the field thetas of V and *thetas*. In the case of a same entry in the field leaders of V and *leaders*, the least common multiple of the corresponding entries of the field thetas of V and *thetas* is taken. The table *leaders* is supposed to be sorted by decreasing order. This function belongs to bad_splitting_tree.c. It is run by 53 tests, including 4 in the bad library and 10 in the DifferentialAlgebra package.

void * bad_scanf_splitting_tree (void *A) [Function]

The parsing function for splitting trees. It is called by ba0_scanf/%splitting_tree. The read splitting tree is verbose. This function belongs to bad_splitting_tree.c. It is not run by any test.

void bad_printf_splitting_tree (void *A) [Function]

The printing function for splitting trees. It is called by ba0_printf/%splitting_tree. This function belongs to bad_splitting_tree.c. It is not run by any test.

void bad_dot_splitting_tree (struct bad_splitting_tree *tree) [Function]

Print *tree* as a directed graph, following the syntax of graphviz/dot. Edges are labelled using the encoding given by bad_typeof_splitting_edge_to_string. The

shapes of the vertices are functions are given by `bad_shapeof_splitting_vertex_to_string`. This function belongs to `bad_splitting_tree.c`. It is run by 7 tests, including 4 in the `bad` library and 0 in the `DifferentialAlgebra` package.

Description of type `bad_splitting_vertex`. This data type is a subtype of `bad_splitting_tree`. It permits to describe one vertex of the tree. Each vertex corresponds to a quadruple / regular chain, which is identified by its *number*.

The field `number` contains the number of the vertex.

The field `is_first` indicates if the vertex is a *first* vertex. First vertices play a special role in elimination methods for they provide bounds which permit to discard quadruples by means of a dimension argument.

The field `edges` contains the table of the edges starting from the vertex towards other vertices of the splitting tree. This table is sorted by increasing `dst` number.

The field `shape` contains the shape of the vertex.

The fields `thetas` and `leaders` are only meaningful if the successors of the vertex in the splitting tree were obtained by a process involving a differential reduction step. In such a case, `leaders` contains the leaders of the regular differential chain used to performed the reduction while `thetas` contains the least common multiple of the derivative operators applied to these regular differential chain elements by the reduction. Both tables have the same size and there is a one-to-one correspondence between there elements.

The field `discarded_branch` indicates if a possible branch, starting from the vertex, was discarded because of the presence of differential inequations. This type declaration belongs to `bad_splitting_vertex.h`.

```
struct bad_splitting_vertex
{
// the number of the vertex which is also the number of the quadruple
  ba0_int_p number;
// indicate if the vertex is a ``first'' vertex
  bool is_first;
// the shape of the vertex
  enum bad_shapeof_splitting_vertex shape;
// the edges towards the children of the vertex
  struct bad_tableof_splitting_edge edges;
// the derivative operators involved in a reduction process (if applicable)
  struct bav_tableof_term thetas;
// the leaders of the polynomials they have applied to (if applicable)
  struct bav_tableof_variable leaders;
};
```

Description of type `bad_shapeof_splitting_vertex`. This data type is a subtype of `bad_splitting_vertex`. It permits to associate a *shape* to a vertex.

- parallelogram vertices correspond to vertices which are rejected by means of a dimension argument
- triangle vertices correspond to vertices which are discarded because a new equation is incompatible with some existing inequation
- box vertices correspond to output regular differential chains
- hexagon vertices correspond to systems handled by the *regCharacteristic* algorithm
- ellipse vertices correspond to any other case.

This type declaration belongs to `bad_splitting_vertex.h`.

```
enum bad_shapeof_splitting_vertex
{
  bad_parallelogram_vertex,
  bad_triangle_vertex,
  bad_box_vertex,
  bad_ellipse_vertex,
  bad_hexagon_vertex
};
```

`void bad_init_splitting_vertex (struct bad_splitting_vertex *V)` [Function]

Initialize V to an empty vertex with number `BAD_NOT_A_NUMBER`. This function belongs to `bad_splitting_vertex.c`. It is run by 145 tests, including 66 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`struct bad_splitting_vertex * bad_new_splitting_vertex (void)` [Function]

Allocate a new vertex, initialize it and return it. This function belongs to `bad_splitting_vertex.c`. It is run by 145 tests, including 66 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_splitting_vertex (struct bad_splitting_vertex *dst, struct bad_splitting_vertex *src)` [Function]

Assign *src* to *dst*. This function belongs to `bad_splitting_vertex.c`. It is not run by any test.

`void bad_merge_thetas_leaders_splitting_vertex (struct bad_splitting_vertex *V, struct bav_tableof_term *thetas, struct bav_tableof_variable *leaders)` [Function]

Merge the field `leaders` of V with *leaders* so that the resulting field `leaders` of V remains sorted by decreasing order. Perform corresponding operations on the field `thetas` of V and *thetas*. In the case of a same entry in the field `leaders` of V and *leaders*, the least common multiple of the corresponding entries of the field `thetas` of V and *thetas* is taken. The table *leaders* is supposed to be sorted by increasing order. This function belongs to `bad_splitting_vertex.c`. It is run by 53 tests, including 4 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`char * bad_shapeof_splitting_vertex_to_string (enum bad_shapeof_splitting_vertex shape)` [Function]

Return a string encoding for *shape*. This function belongs to `bad_splitting_vertex.c`. It is run by 7 tests, including 4 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void * bad_scanf_splitting_vertex (void *A)` [Function]

The general parsing function for vertices. It can be called through `ba0_scanf/%splitting_vertex`. The expected syntax is as follows (the last string is supposed to be `true` or `false`):

```
<%d, is_first, shape, %t[%splitting_edge], %t[%term], %t[%v]>
```

This function belongs to `bad_splitting_vertex.c`. It is not run by any test.

`void bad_printf_splitting_vertex (void *A)` [Function]

The general printing function for vertices. It can be called through `ba0_printf/%splitting_vertex`. This function belongs to `bad_splitting_vertex.c`. It is not run by any test.

Description of type `bad_splitting_edge`. This data type is a subtype of `bad_splitting_vertex`. It permits to describe an edge of the splitting tree generated by a differential elimination algorithm.

The fields `src` and `dst` contain the *numbers* of the vertices connected by the edge.

For some values of the `type` field, the leader of the involved polynomial is stored in the `leader` field. Similarly, for some values of the `type` field, a multiplicity information is stored in the `multiplicity` field. This type declaration belongs to `bad_splitting_edge.h`.

```
struct bad_splitting_edge
{
// the type of the edge
  enum bad_typeof_splitting_edge type;
// the number of the source vertex
  ba0_int_p src;
// the number of the target vertex
  ba0_int_p dst;
// the leader of the differential polynomial (bad_separant_edge)
  struct bav_variable *leader;
// the multiplicity of the factor (bad_separant_edge)
  ba0_int_p multiplicity;
};
```

Description of type `bad_typeof_splitting_edge`. This data type is a subtype of `bad_splitting_edge`. It permits to associate a type to an edge in the splitting tree generated by `bad_Rosenfeld_Groebner`. This type declaration belongs to `bad_splitting_edge.h`.

```
enum bad_typeof_splitting_edge
{
// not an edge
  bad_none_edge,
// critical pair leading to a non trivial differential polynomial
  bad_critical_pair_edge,
  bad_critical_pair_novar_edge, // result is a base field element
// reduction to zero
  bad_redzero_edge,
// splittings introducing an inequation at each edge
  bad_first_edge,               // processing the initial equations
  bad_factor_edge,              // plain factorization
  bad_initial_edge,             // initial
  bad_separant_edge,            // separant
// splittings not introducing inequations
  bad_regularize_edge,          // complete
  bad_reg_characteristic_edge   // reg_characteristic
};
```

`char * bad_typeof_splitting_edge_to_string (enum bad_typeof_splitting_edge type)` [Function]

Return a string encoding for *type*. The encoding is given by the following table

`"xxx"` `bad_none_edge`
`"cri"` `bad_critical_pair_edge`

```
"crik" bad_critical_pair_novar_edge
"rdz"  bad_redzero_edge
"frst" bad_first_edge
"fact" bad_factor_edge
"ini"  bad_initial_edge
"sep"  bad_separant_edge
"regu" bad_regularize_edge
"regc" bad_reg_characteristic_edge
```

This function belongs to `bad_splitting_edge.c`. It is run by 7 tests, including 4 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`struct bav_symbol * bad_leader_symbol_splitting_edge (struct bad_splitting_edge *E)` [Function]

Return the symbol of the `leader` field of *E*. Exception `BA0_ERRALG` is raised if the `leader` field of *E* is `BAV_NOT_A_SYMBOL`. This function belongs to `bad_splitting_edge.c`. It is run by 7 tests, including 4 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`bool bad_has_var_typeof_splitting_edge (enum bad_typeof_splitting_edge type)` [Function]

Return `true` if the `leader` field should be different from `BAV_NOT_A_VARIABLE` for an edge of type *type*. This function belongs to `bad_splitting_edge.c`. It is run by 145 tests, including 66 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`bool bad_has_multiplicity_typeof_splitting_edge (enum bad_typeof_splitting_edge type)` [Function]

Return `true` if the `mult` field is meaningful for an edge of type *type*. This function belongs to `bad_splitting_edge.c`. It is run by 7 tests, including 4 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_set_tsdvm_splitting_edge (struct bad_splitting_edge *E, enum bad_typeof_splitting_edge type, ba0_int_p src, ba0_int_p dst, struct bav_variable *leader, ba0_int_p multiplicity)` [Function]

Assign *type*, *src* and *dst* to the corresponding fields of *E*. Store *src_var* and *dst_var* in the two first entries of the field `leaders` of *E*. This function belongs to `bad_splitting_edge.c`. It is run by 53 tests, including 4 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_init_splitting_edge (struct bad_splitting_edge *E)` [Function]

Initialize *E*. This function belongs to `bad_splitting_edge.c`. It is run by 53 tests, including 4 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`struct bad_splitting_edge * bad_new_splitting_edge (void)` [Function]

Allocate a new edge, initialize it and return it. This function belongs to `bad_splitting_edge.c`. It is run by 53 tests, including 4 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_set_splitting_edge (struct bad_splitting_edge *E, struct bad_splitting_edge *F)` [Function]

Assign F to E. This function belongs to `bad_splitting_edge.c`. It is not run by any test.

`void * bad_scanf_splitting_edge (void *A)` [Function]

The parsing function for splitting edges. It is called by `ba0_scanf/%splitting_edge`. The expected syntax is `<type, src, dst>` or `<type, src, dst, leader, multiplicity>`. This function belongs to `bad_splitting_edge.c`. It is not run by any test.

`void bad_printf_splitting_edge (void *A)` [Function]

The printing function for splitting edges. It is called by `ba0_printf/%splitting_edge`. This function belongs to `bad_splitting_edge.c`. It is not run by any test.

8.15 Splitting control

Description of type `bad_typeof_dimension_lower_bound`. This data type permits to specify to differential elimination algorithms the type of dimension lower bound to be applied. Such a lower bound permits to cut any branch of the splitting tree leading to regular chains of differential dimension strictly less than the number of input equations. Note that such a strategy is proved in the algebraic case and in the case of a single differential equation. The other cases are conjectural. This type declaration belongs to `bad_splitting_control.h`.

```
enum bad_typeof_dimension_lower_bound
{
// do not take into account any dimension lower bound
  bad_no_dimension_lower_bound,
// apply it in the case of a non-differential system
  bad_algebraic_dimension_lower_bound,
// apply it in the case of an ordinary differential system
  bad_ode_dimension_lower_bound,
// apply it in the case of a partial differential system
  bad_pde_dimension_lower_bound
};
```

Description of type `bad_splitting_control`. This data type permits to specify a splitting control strategy to differential elimination algorithms.

The field `first_leaf_only` indicates if the differential elimination process must stop at the first consistent regular chain obtained. Default value is `false`.

The field `dimlb` indicates the type of dimension argument that should be taken into account. Default value is `bad_algebraic_dimension_lower_bound`.

The field `apply_dimlb_one_eq` indicates if the case of a single input equation should be considered specifically (provided that `dimlb` is different from `bad_no_dimension_lower_bound`). If set to `true`, branches of the splitting tree leading to regular chains of differential dimension strictly less than the number of input equations (which is then equal to 1) are discarded. Default value is `true`.

The field `DenefLipshitz` indicates if we are performing the differential elimination stage of `bas_DenefLipshitz`. If this is the case, the other `first_leaf_only` and `apply_dimlb_one_eq` are set to `false` and `dimlb` is set to `bad_no_dimension_lower_bound`. Moreover,

the inequations are not used to discard quadruples before the final algebraic processing. This type declaration belongs to `bad_splitting_control.h`.

```
struct bad_splitting_control
{
// Stop at the first consistent regular chain (if any)
  bool first_leaf_only;
// Should a dimension lower bound be taken into account ?
  enum bad_typeof_dimension_lower_bound dimlb;
// Should the case of a single input equation be handled specifically?
  bool apply_dimlb_one_eq;
// Are we in the differential elimination stage of the DenefLipshitz algorithm?
  bool DenefLipshitz;
};
```

`void bad_init_splitting_control (struct bad_splitting_control *S)` [Function]

Initialize C. The fields `first_leaf_only` and `DenefLipshitz` are set to `false`. The fields `dimlb` and `apply_dimlb_one_eq` are set to `bad_algebraic_dimension_lower_bound` and `true`. This function belongs to `bad_splitting_control.c`. It is run by 147 tests, including 68 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`struct bad_splitting_control * bad_new_splitting_control (void)` [Function]

Allocate a new `struct bad_splitting_control *`, initialize it and return it. This function belongs to `bad_splitting_control.c`. It is run by 7 tests, including 7 in the `bad` library and 0 in the `DifferentialAlgebra` package.

`void bad_set_splitting_control (struct bad_splitting_control *S, struct bad_splitting_control *T)` [Function]

Assign T to S. This function belongs to `bad_splitting_control.c`. It is not run by any test.

`void bad_set_DenefLipshitz_splitting_control (struct bad_splitting_control *S, bool b)` [Function]

Assign b to the field `DenefLipshitz` of C. If b is `true` then `first_leaf_only` and `apply_dimlb_one_eq` are set to `false` and `dimlb` is set to `bad_no_dimension_lower_bound`. This function belongs to `bad_splitting_control.c`. It is run by 49 tests, including 0 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_set_first_leaf_only_splitting_control (struct bad_splitting_control *S, bool b)` [Function]

Assign b to the field `first_leaf_only` of C. This function belongs to `bad_splitting_control.c`. It is run by 65 tests, including 16 in the `bad` library and 10 in the `DifferentialAlgebra` package.

`void bad_set_dimension_lower_bound_splitting_control (struct bad_splitting_control *S, enum bad_typeof_dimension_lower_bound lb, bool one_eq)` [Function]

Assign *lb* and *one_eq* to the fields `dimlb` and `apply_dimlb_one_eq` of C. This function belongs to `bad_splitting_control.c`. It is run by 69 tests, including 0 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`bool bad_apply_dimension_lower_bound_splitting_control` [Function]
`(struct bad_splitting_control *S, struct bad_regchain *C, struct bap_listof_polynom_mpz *eqns, struct bad_base_field *K, bool differential, ba0_int_p *numberof_input_equations)`

Return `true` if differential elimination methods must discard any output regular differential chain involving more than the number of input equations.

The decision depends on S, the number of input equations, and the number of derivations involved in our problem and whether the elimination process is differential or not (information in *differential*).

The number of input equations is defined as the size of C plus the size of *eqns* plus the number of implicit equations associated to the parameters occurring in their polynomials which are not leaders. This number is returned in **numberof_input_equations*. This function belongs to `bad_splitting_control.c`. It is run by 145 tests, including 66 in the `bad` library and 30 in the `DifferentialAlgebra` package.

8.16 Quadruples

Description of type `bad_quadruple`. This data type implements quadruples $G = \langle A, D, P, S\rangle$, which are processed by the RosenfeldGroebner algorithm.

The field `A` contains a regular differential chain.

The field `D` contains the list of the critical pairs to be processed.

The field `P` contains the list of the differential polynomials to be processed.

The field `S` contains a list of inequations.

Quadruples are identified in splitting trees by a *number* which is held by the `number` field of `A`.

The RosenfeldGroebner algorithm starts with $G = \langle \emptyset, \emptyset, P_0, S_0\rangle$ where P_0 and S_0 are the input lists of equations and inequations. It computes a finite set of quadruples containing regular systems i.e. of the form $G = \langle A, \emptyset, \emptyset, S\rangle$ and S partially reduced with respect to A. These regular systems are then processed by the regCharacteristic algorithm to produce a finite set of regular differential chains. This type declaration belongs to `bad_quadruple.h`.

```
struct bad_quadruple
{
// the (somehow) already processed differential polynomials
  struct bad_regchain A;
// the list of critical pairs waiting to be processed
  struct bad_listof_critical_pair *D;
// the list of differential polynomials waiting to be processed
  struct bap_listof_polynom_mpz *P;
// the list of inequations (different from zero)
  struct bap_listof_polynom_mpz *S;
};
```

`void bad_init_quadruple (struct bad_quadruple *G)` [Function]

Initialize G. This function belongs to `bad_quadruple.c`. It is run by 156 tests, including 77 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`struct bad_quadruple * bad_new_quadruple (void)` [Function]
Allocate a new quadruple, initialize it and return it. This function belongs to `bad_quadruple.c`. It is run by 156 tests, including 77 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_quadruple (struct bad_quadruple *P, struct bad_quadruple *Q)` [Function]
Assign Q to P. This function belongs to `bad_quadruple.c`. It is not run by any test.

`ba0_int_p bad_get_number_quadruple (struct bad_quadruple *G)` [Function]
Return the number of G. This function belongs to `bad_quadruple.c`. It is run by 145 tests, including 66 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_number_quadruple (struct bad_quadruple *G, ba0_int_p number)` [Function]
Set the number of G to *number*. This function belongs to `bad_quadruple.c`. It is run by 154 tests, including 75 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_set_next_number_quadruple (struct bad_quadruple *G, struct bad_splitting_tree *tree)` [Function]
Set the number of G to the next number obtained from *tree*. This function belongs to `bad_quadruple.c`. It is run by 154 tests, including 75 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_mark_indets_quadruple (struct bav_dictionary_variable *dict, struct bav_tableof_variable *vars, struct bad_quadruple *Q)` [Function]
Append to *vars* the variables occurring in Q which are not already present in *vars*. Every element of *vars* is supposed to be registered in the dictionary *dict*. This function belongs to `bad_quadruple.c`. It is not run by any test.

`void bad_extend_quadruple_regchain (struct bad_quadruple *Q, struct bad_regchain *C, struct bad_base_field *K)` [Function]
Extend the field `A` of Q with C (see `bad_extend_regchain`). Insert the initials of C in the field `S` of Q. Insert the separants also if the field `A` of Q holds the squarefree property. This function belongs to `bad_quadruple.c`. It is run by 147 tests, including 68 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_insert_in_P_quadruple (struct bad_quadruple *P, struct bad_quadruple *Q, struct bap_polynom_mpz *poly)` [Function]
Assign Q to P. Then insert *poly* in the field `P` of P. This function belongs to `bad_quadruple.c`. It is run by 119 tests, including 57 in the `bad` library and 29 in the `DifferentialAlgebra` package.

`void bad_insert_in_S_quadruple (struct bad_quadruple *P, struct bad_quadruple *Q, struct bap_polynom_mpz *poly)` [Function]
Assign Q to P. Then insert *poly* in the field `S` of P. This function belongs to `bad_quadruple.c`. It is run by 119 tests, including 57 in the `bad` library and 29 in the `DifferentialAlgebra` package.

`struct bap_listof_polynom_mpz *` [Function]
`bad_insert_in_listof_polynom_mpz (struct bap_polynom_mpz *p, struct bap_listof_polynom_mpz *L)`

Insert a copy of p in L. In place function. The list L is assumed to be the component P or S of some quadruple. If p is already present in L, it is not duplicated. The systematic use of this function keeps lists sorted. This function belongs to `bad_quadruple.c`. It is run by 155 tests, including 76 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`struct bap_listof_polynom_mpz *` [Function]
`bad_delete_from_listof_polynom_mpz (struct bap_polynom_mpz *p, struct bap_listof_polynom_mpz *L, bool *deletion)`

Remove p from the list L. The argument *deletion* is set to `true` if p was present in L, else it is set to `false`. It may be the zero pointer. This function belongs to `bad_quadruple.c`. It is not run by any test.

`void bad_preprocess_equation_quadruple (struct` [Function]
`bap_product_mpz *A, struct bap_tableof_polynom_mpz *ineqs, struct bap_tableof_product_mpz *factored_ineqs, bool *discarded_branch, struct bad_quadruple *G, struct bad_base_field *K)`

Simplify A using the inequations of G by means of gcd computations. Factors of A which belong to the base field K or which divide some inequation of G are discarded. This simplification process may exhibit a factorization of some inequation of G. Any such factorization is recorded as an entry in *ineqs* and *factored_ineqs*. Factors of inequations which cannot vanish are also discarded: the nonzero elements of the base field K. If nonzero, the argument *discarded_branch* is set to `true` if at least one factor which is not a base field element is discarded. This function belongs to `bad_quadruple.c`. It is run by 94 tests, including 35 in the `bad` library and 13 in the `DifferentialAlgebra` package.

`void bad_report_simplification_of_inequations_quadruple` [Function]
`(struct bad_tableof_quadruple *tabG, struct bap_tableof_polynom_mpz *ineqs, struct bap_tableof_product_mpz *factored_ineqs)`

Replace in the field `S` of each quadruple of *tabG* any occurrence of any polynomial of *ineqs* by its factors, given in *factored_ineqs*. These factors are supposed to be processed by `bad_preprocess_equation_quadruple`. This function belongs to `bad_quadruple.c`. It is run by 11 tests, including 9 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`bool bad_simplify_and_store_in_P_quadruple (struct` [Function]
`bad_quadruple *G, bool *discarded_branch, struct bap_polynom_mpz *p, struct bad_base_field *K)`

Store in the `P` field of G the polynomial p after the simplifications performed by `bad_preprocess_equation_quadruple`. Return `false` if p is proved to be nonzero so that G becomes inconsistent else return `false`. If nonzero, the argument *discarded_branch* is set to `true` if at least one factor which is not a base field element is discarded. This

function belongs to `bad_quadruple.c`. It is run by 2 tests, including 1 in the `bad` library and 1 in the `DifferentialAlgebra` package.

`bool bad_simplify_and_store_in_S_quadruple (struct bad_quadruple *G, bool *discarded_branch, struct bap_polynom_mpz *p, struct bad_base_field *K)` [Function]

Store in the `S` field of G the polynomial p after the simplifications performed by `bad_preprocess_equation_quadruple`. Return `false` if p is proved to be zero so that G becomes inconsistent else return `false`. If nonzero, the argument *discarded_branch* is set to `true` if at least one factor which is not a base field element is discarded. This function belongs to `bad_quadruple.c`. It is run by 93 tests, including 34 in the `bad` library and 13 in the `DifferentialAlgebra` package.

`void bad_split_on_factors_of_equations_quadruple (struct bad_tableof_quadruple *tabG, struct bad_splitting_tree *tree, struct bap_product_mpz *prod, struct ba0_tableof_bool *keep, struct bap_polynom_mpz *uctum)` [Function]

Generate in *tabG* the quadruples obtained from the top quadruple G of *tabG*, covering the cases where *prod* and *uctum* both vanish.

To each factor of *prod*, there exists a corresponding entry of *keep*. The factors of *prod* whose corresponding entry in *keep* is `false` are not considered.

Denote $a_1 \cdots a_n$ the considered factors of *prod*

If *uctum* is not `BAP_NOT_A_POLYNOMIAL_mpz` then n new quadruples Q_i equal to $G, a_i = 0, a_1 \cdots a_{i-1} \neq 0, uctum = 0$. Moreover, the quadruple G is augmented with $a_1 \cdots a_n \neq 0$.

If *uctum* is `BAP_NOT_A_POLYNOMIAL_mpz` then $n - 1$ new quadruples Q_i equal to $G, a_i = 0, a_1 \cdots a_{i-1} \neq 0$ are generated. Moreover, the quadruple G is augmented with $a_1 \cdots a_{n-1} \neq 0$.

In quadruples, new equations are stored in the field `P`. New inequations are stored in the field `S`.

The argument *tree* is used to assign a *number* to each new quadruple Q_i. These numbers are equal to i plus some offset. The number of G is not modified. This function belongs to `bad_quadruple.c`. It is run by 139 tests, including 64 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_pick_and_remove_quadruple (struct bap_polynom_mpz *p, struct bad_quadruple *G, struct bad_critical_pair * *pair, struct bad_selection_strategy *S)` [Function]

It is assumed that one of the components P or D of the quadruple is non empty. Assign to p the first element of P or the Δ-polynomial defined by the first critical pair of D. The choice is defined by *strategy*. The picked element is removed from the list. If p is picked from P then zero is assigned to *pair* else the picked pair is assigned to *pair*. The pointer *pair* is allowed to be zero. This function belongs to `bad_quadruple.c`. It is run by 154 tests, including 75 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_reg_characteristic_quadruple (struct bad_intersectof_regchain *tabC, struct bad_quadruple *G, struct bad_regchain *ideal, struct bad_base_field *K)` [Function]

Denote I the unit ideal if *ideal* is zero and the ideal defined by *ideal* otherwise. It is assumed that $G = \langle A, \emptyset, \emptyset, S\rangle$, that $A = 0,\ S \neq 0$ is a regular differential system and that $[A] : S^\infty \subset I$. This function stores in *tabC* a list of regular differential chains $A_1, \ldots, A_t$ such that $[A] : S^\infty \subset [A_1] : {H_{A_1}}^\infty \cap \cdots \cap [A_t] : {H_{A_t}}^\infty \subset I$. In the case I is prime the function only returns one regular chain. If moreover I is a minimal prime of $[C] : {H_C}^\infty$ then the returned regular chain is a characteristic set of I. Polynomials are supposed to have coefficients in K. This function belongs to `bad_quadruple.c`. It is run by 149 tests, including 71 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_complete_quadruple (struct bad_tableof_quadruple *tabG, struct bav_tableof_term *theta, bool *discarded_branch, struct bad_splitting_tree *tree, struct bap_polynom_mpz *r, struct bad_regchain *ideal, struct bad_base_field *K, struct bad_selection_strategy *strategy)` [Function]

Denote $G = \langle A, D, P, S\rangle$ the last element of *tabG*.

In the nondifferential context, store r in A and possibly a critical pair in D if A contains an element with the same leader as r.

In the differential context, it is assumed that r is partially reduced with respect to A. The function stores r in A. Some critical pairs are generated and stored in D. This list is sorted according to *strategy*. The elements of S are reduced partially with respect to A.

Both contexts.

The quenching operation, applied on A may split A in finitely many different regular chains. In this case, the quadruple G is split into the same number of quadruples. The resulting quadruples are stored in *tabG*, at indices greater than or equal to that of G (the functions reuses the input quadruple G).

The parameter *ideal* is provided to the quenching operation and permits to avoid some splittings: if an element of A factors as $f_1\, f_2 = 0$ then the branch $f_i = 0$ is generated only if f_i is zero or a zero divisor modulo *ideal*.

The initial and the separants of r are not stored in S. Polynomials are supposed to have coefficients in K.

If nonzero, the argument *discarded_branch* is set to `true` if at least one branch of the splitting tree is discarded because of the presence of inequations. This may happen either in the quenching process and in the partial reduction process of the inequations. This function belongs to `bad_quadruple.c`. It is run by 148 tests, including 73 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void * bad_scanf_quadruple (void *A)` [Function]

The parsing function for quadruples. It is called by `ba0_scanf/%quadruple`. The expected format is `quadruple %d (regular chain, D = list of critical pairs, P = list of polynomials, S = list of polynomials)`. The quadruple number may be omitted. This function belongs to `bad_quadruple.c`. It is not run by any test.

`void bad_printf_quadruple (void *A)` [Function]

The printing function for quadruples. It is called by `ba0_printf/%quadruple`. This function belongs to `bad_quadruple.c`. It is not run by any test.

8.17 Differential elimination methods

`void bad_Rosenfeld_Groebner (struct bad_intersectof_regchain *ideal, struct bad_splitting_tree *T, struct bap_tableof_polynom_mpz *P, struct bap_tableof_polynom_mpz *S, struct bad_base_field *K0, struct bad_regchain *A0, struct bad_splitting_control *C)` [Function]

Apply a version of the *RosenfeldGrobner* algorithm.

This version handles as well differential systems as non differential ones. For a better legibility, this documentation assumes that the system is differential.

Assign to *ideal* a set of differential regular chains. These differential regular chains define differential ideals whose intersection is equal to the radical of the differential ideal $[P \cup A_0] : (S \cup H_{A_0})^\infty$.

The argument *A0* provides a starting regular differential chain. If it is nonempty then all computed regular differential chains extend it.

The function `bad_fast_primality_test_regchain` is applied over each component of *ideal*. Thus some output regular differential chains may have the prime ideal property set.

The intersection may involve redundant components. The function `bad_remove_redundant_components_intersectof_regchain`, which may detect and remove some of them is not called.

The argument S may be zero.

If different from zero, *K0* contains a description of the base field of the differential polynomials. In this case, every nonzero polynomial p lying in the base field is considered as nonzero by *RosenfeldGroebner*: the case $p = 0$ is not discussed. If *K0* involves a non trivial regular differential chain then the elements of this chain are incorporated to each output regular differential chain. If *K0* is zero then the base field is supposed to be the purely transcendental extension of the field of the rational numbers with the independent variables.

The properties of *ideal* are supposed to be set by the calling function. Exception `BAD_ERRIBF` is raised if the properties of the regular chain of *K0* are not compatible with those of *ideal*. Exceptions `BAD_ERRIAC` or `BAD_ERRCRC` are raised if the regular chain *A0* is not compatible with the one of *K0* or with the properties of *ideal*.

The argument C, which may be zero, permits to control the splitting mechanism of *RosenfeldGroebner*.

The argument T, which may be zero, permits to compute and return a description of the splitting tree implicitly generated by *RosenfeldGroebner*. When *RosenfeldGroebner* is used to perform the differential elimination stage of the *DenefLipshitz* algorithm, the field `DenefLipshitz` of C should be set to `true` and the activity of *tree* should be either `bad_quiet_splitting_tree` or `bad_verbose_splitting_tree`. This function belongs to `bad_Rosenfeld_Groebner.c`. It is run

by 147 tests, including 68 in the `bad` library and 30 in the `DifferentialAlgebra` package.

`void bad_pardi (struct bad_regchain *Cbar, bav_Iordering rbar, struct bad_regchain *C)` [Function]

Compute a regular differential chain by change of ranking. Assign to *Cbar* a regular chain with respect to the ordering *rbar* defining the same ideal as *C*. The differential ideal and prime ideal properties of *Cbar* are inherited from *C*. The other properties must be set before calling the function. This function belongs to `bad_pardi.c`. It is run by 11 tests, including 9 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_first_quadruple (struct bad_tableof_quadruple *tabG, struct bad_attchain *attrib, struct bap_tableof_polynom_mpz *P, struct bap_tableof_polynom_mpz *S, enum bad_typeof_reduction type_red, struct bad_base_field *K, struct bad_regchain *A)` [Function]

Initialize the field `A` of the first quadruple of *tabG* with the defining equations of *K*, extended with the polynomials of *A*. Initialize the field `P` with *P*. Initialize the field `S` with *S* and the initials of *A* (plus the separants if *attrib* holds the squarefree property). This function belongs to `bad_Rosenfeld_Groebner.c`. It is run by 147 tests, including 68 in the `bad` library and 30 in the `DifferentialAlgebra` package.

8.18 The Low Power Theorem

The Low Power Theorem was first proven by Ritt and generalized by Kolchin. It permits to compute a representation of the radical of the differential ideal generated by a single differential polynomial as an irredundant intersection of regular differential chains. Computationnally speaking, its main step consists in computing a preparation equation (Kolchin, IV, 13) of a differential polynomial, w.r.t. a regular differential chain.

Given a differential polynomial F and a regular differential chain $A = A_1, \ldots, A_r$, a representation equation has the form $HF = \text{sum } c_i t_i$ modulo $[z_k = A_k]$. The coefficients c_i are differential polynomials reduced and regular w.r.t A. The terms t_i are power products of derivatives of z_k, each z_k standing for some element A_k of the chain A. The terms t_i are ordered by increasing total degree. The differential polynomial H is a power product of initials and separants of the chain A.

The base field K provides a regular chain $A_1, \ldots, A_s$, which occur at the bottom of A, and should be considered as base field equations. Reductions by these elements should not be taken into account while applying the Low Power Theorem: the terms t_i which involve at least one derivative of any z_k with $k \leq s$ are moved to the end of the table `terms`.

Description of type `bad_preparation_term`. This data type implements one term in a preparation equation. Denote $A = A_1, \ldots, A_r$ a differential regular chain. Introduce r differential indeterminates $z_1, \ldots, z_r$. A `bad_preparation_term` represents a sequence of terms on the derivatives of the z_i. All fields have the same size. This type declaration belongs to `bad_low_power_theorem.h`.

```
struct bad_preparation_term
{
// z[i] = an index in the range [0, ..., r-1]
```

```
  struct ba0_tableof_int_p z;
// theta[i] = a power product of derivations = a derivation operator
  struct bav_tableof_term theta;
// deg[i] = a degree
  struct ba0_tableof_int_p deg;
};
```

Description of type `bad_preparation_equation`. This data type implements preparation equations (Kolchin, IV, 13). Given F and $A = A_1, \ldots, A_r$ a preparation equation mostly represents $H\,F$ as the sum of the pairwise products of the coefficients by the terms modulo $(z_i = A_i)$. This type declaration belongs to `bad_low_power_theorem.h`.

```
struct bad_preparation_equation
{
// power product of initials and separants
  struct bap_product_mpz H;
// the coefficients (reduced and regular w.r.t. A)
  struct bap_tableof_polynom_mpz coeffs;
// the terms
  struct bad_tableof_preparation_term terms;
// the polynomial for which the preparation equation is defined
  struct bap_polynom_mpz *F;
// the denominator of a rational number (to handle
// polynomials with rational number coefficients)
  ba0__mpz_struct *denom;
// the regular chain A
  struct bad_regchain *A;
// the base field - its elements are on the bottom of A
  struct bad_base_field *K;
};
```

`void bad_set_settings_preparation (char *zstring)` [Function]

Define as *zstring* the format which is used for printing the z_i. If this parameter is zero, the format is restored to its default value: `z%d`. This function belongs to `bad_low_power_theorem.c`. It is run by 65 tests, including 0 in the `bad` library and 65 in the `DifferentialAlgebra` package.

`void bad_get_settings_preparation (char **zstring)` [Function]

Assign to *zstring* the current format identifier used for printing the z_i. The argument may be zero. This function belongs to `bad_low_power_theorem.c`. It is not run by any test.

`void bad_init_preparation_equation (struct bad_preparation_equation *E)` [Function]

Initialize E. This function belongs to `bad_low_power_theorem.c`. It is run by 10 tests, including 8 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`struct bad_preparation_equation * bad_new_preparation_equation (void)` [Function]

Allocate a new preparation equation, initialize it and return it. This function belongs to `bad_low_power_theorem.c`. It is not run by any test.

`void bad_set_preparation_equation_polynom (struct bad_preparation_equation *E, struct bap_polynom_mpz *F, ba0_mpz_t denom, struct bad_regchain *A, struct bad_base_field *K, struct bap_polynom_mpz * *ddz)` [Function]

Assign to E a preparation equation for $F/denom$ and A. The regularity test on the coefficients c_i may raise an exception. The zero divisor is then recovered in *ddz*. The parameter K provides the base field and indicates the elements of A which must be considered as base field equations. This function belongs to `bad_low_power_theorem.c`. It is run by 10 tests, including 8 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_preparation_congruence (ba0_int_p *l, bav_Idegree *q, struct bad_preparation_equation *E)` [Function]

Assign to q the minimum total degree of the power products of derivatives of the z_i which occur in E. Assign to l the number of such terms which have degree q. Observe that these l monomials occur at the first place of the tables of E. Terms which involve derivatives of z_i corresponding to base field equations cannot be part of the congruence. This function belongs to `bad_low_power_theorem.c`. It is run by 6 tests, including 4 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`bool bad_low_power_theorem_condition_to_be_a_component (struct bad_preparation_equation *E)` [Function]

Return `true` if the preparation equation (congruence) E satisfies the necessary and sufficient condition provided in the Low Power Theorem (Kolchin, IV, 15, Theorem 6) for A to be an irredundant component of the representation of the radical of the differential ideal generated by F as an intersection of regular differential chains.

Observe that the function does not check the fact that A only consists of one element i.e. that $A = A_1$. This condition is not checked here in order to permit the application of the Low Power Theorem to differential polynomials with coefficients taken in some nontrivial base field. This function belongs to `bad_low_power_theorem.c`. It is run by 6 tests, including 4 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_low_power_theorem_simplify_intersectof_regchain (struct bad_intersectof_regchain *S0, struct bad_intersectof_regchain *T0, struct bad_base_field *K)` [Function]

The intersection T is assumed to be produced by `bad_Rosenfeld_Groebner` over a single differential polynomial F, with coefficients in the base field K. Remove from T the redundant components of the radical of the differential ideal generated by F, by applying the Low Power Theorem. The result is stored in S. The result may involve components which are not in T. It may even involve more components than T. This function belongs to `bad_low_power_theorem.c`. It is run by 6 tests, including 4 in the `bad` library and 2 in the `DifferentialAlgebra` package.

`void bad_printf_preparation_equation (void *A)` [Function]

The printing function for preparation equations. It is called by `ba0_printf/%preparation_equation`. This function belongs to `bad_low_power_theorem.c`. It is run by 5 tests, including 4 in the `bad` library and 1 in the `DifferentialAlgebra` package.

9 The bas library

9.1 Introduction

The `bas` library implements algorithms dedicated to the existence problem of formal power series solutions of differential polynomial systems.

Read the documentation of the `ba0` library about settings.

`void bas_reset_all_settings (void)` [Function]
Reset all settings variables to their default values. This function must be called outside sequences of calls to the library. This function belongs to `bas_common.c`. It is run by 65 tests, including 0 in the `bas` library and 65 in the `DifferentialAlgebra` package.

9.1.1 Basic types, constants and naming conventions

The name of every exported object defined in the library starts by `bas_`. Functions starting by `bas_init` initialize raw data structures (constructors). Functions starting by `bas_new` dynamically allocate data structures and apply the corresponding `bas_init` functions on them. Functions starting by `bas_reset` reset data structures which are assumed to be already initialized. Functions starting by `bas_set` perform assignments and systematically copy the sources to the destinations.

A *sequence of calls* to the `bas` library starts with a call to `bas_restart` and finishes by a call to `bas_terminate`.

`void bas_restart (ba0_int_p` *`time_limit`*`, ba0_int_p` *`memory_limit`*`)` [Function]
Call the `bas_restart` function with the same parameters This function belongs to `bas_common.c`. It is run by 156 tests, including 57 in the `bas` library and 65 in the `DifferentialAlgebra` package.

`void bas_terminate (enum ba0_restart_level` *`level`*`)` [Function]
Call the `bas_terminate` function with the same parameter. This function belongs to `bas_common.c`. It is run by 156 tests, including 57 in the `bas` library and 65 in the `DifferentialAlgebra` package.

9.2 Global variables and settings

None are defined.

9.3 Bibliography

- [BLV25] François Boulier, François Lemaire and Thi Xuân Vu. *The DenefLipshitz Algorithm: Deciding the Existence of Formal Power Series Solutions in Differential Algebra*. `https://hal.science/hal-05294349`. 2025
- [DL84] Jan Denef and Leonard Lipshitz. *Power Series Solutions of Algebraic Differential Equations*. Math. Ann. 267, pages 213-238. 1984

9.4 Leaf contexts

Description of type `bas_leaf_context`. The data type is used by the `bas_Denef_Lipshitz` algorithm. With the vocabulary of [BLV25], it contains the data which is common to all the tasks produced by a single call to Algorithm *DenefLipshitz_leaf*.

The following entries are set once per call to `bas_Denef_Lipshitz`.

The field `smap` permits to associate derivatives such as `y`,`y[x]`, `z[x,x]` to subscripted variables such as `y[0]`, `y[1]`, `z[2]`.

In particular, its subfield `Y` contains the table of the differential indeterminates concerned by the mapping i.e. the ones for which formal power series are sought. This table is sorted by increasing order with respect to the ranking.

Apart `binomials`, the other fields which are tables all have the same size as `Y`. The ith entry of any such table applies to the ith differential indeterminate i.e. to the ith entry of `smap.Y`. Some of these entries may remain unused: the ones for which the differential indeterminate has no defining equation.

The field `point` is an evaluation point defined by the field `smap.Ybar`.

The field `S` is the list of the inequations provided by the user and which apply to constant polynomials with respect to the derivation. These inequations, which include in particular constraints on the subscripted variables associated to differential indeterminates, are provided to the regular chain decomposition method.

The field `q` contains the variable to be used in the context of Hurwitz formula. It stands for a number of differentiations.

The following fields are set once per call to `bas_Denef_Lipshitz_leaf`.

Let i be the index of some y in `smap.Y`. Let s denote the separant of the y defining ODE. Then `kappa.tab[i]` contains an upper bound on the valuation of the series obtained by evaluating s at any formal power series solution of the task. See the field `k` of the `struct bas_task` data structure. It is equal to -1 if y has no defining ODE.

Let i be the index of some y in `smap.Y`. Let f denote the defining ODE of y. Then `ode.tab[i]` is a table containing derivatives of f. If y has no defining ODE then `ode.tab[i]` is empty else it contains at least f.

Let i be the index of some y in `smap.Y`. Let s denote the separant of the defining ODE of y. Then `sep.tab[i]` is a table containing derivatives of s. If y has no defining ODE then `sep.tab[i]` is empty else it contains at least s.

Let i be the index of some y in `smap.Y`. Let f denote the defining ODE of y. The field `order.tab[i]` contains the order of the leader of f.

The field `mumax` contains tables of prolongation orders applied to the defining equations. These tables are used for computing a bound, as defined in [BLV25], returned by the `bas_Denef_Lipshitz_leaf` function.

The field `binomials` contains binomial polynomials involved in the computation process of the polynomials $A(q)$. This field gets regularly updated during the execution of `bas_Denef_Lipshitz_leaf`. This type declaration belongs to `bas_leaf_context.h`.

```
struct bas_leaf_context
{
  struct baz_series_mapping smap;
  struct baz_point_ratfrac point;
```

```
  struct ba0_tableof_int_p kappa;

  struct bap_tableof_tableof_polynom_mpz ode;
  struct ba0_tableof_int_p order;

  struct bap_tableof_tableof_polynom_mpz sep;

  struct bap_listof_polynom_mpz *S;

  struct ba0_tableof_tableof_int_p mumax;

  struct bap_tableof_polynom_mpq binomials;

  struct bav_variable *q;
};
```

`void bas_init_leaf_context (struct bas_leaf_context *U)` [Function]
Initialize U to the empty leaf context. This function belongs to `bas_leaf_context.c`. It is run by 63 tests, including 53 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`struct bas_leaf_context * bas_new_leaf_context (void)` [Function]
Allocate a new leaf context, initialize it and return it. This function belongs to `bas_leaf_context.c`. It is not run by any test.

`void bas_set_leaf_context (struct bas_leaf_context *dst, struct bas_leaf_context *src)` [Function]
Assign *src* to *dst*. This function belongs to `bas_leaf_context.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`bool bas_has_a_defining_ODE (struct bas_leaf_context *U, ba0_int_p i, struct bap_polynom_mpz **f)` [Function]
Return `true` if the ith differential indeterminate of the field `Y` of U has a defining ODE, else `false`. In the first case, f is assigned a pointer to the ODE. The argument f is allowed to be the zero pointer. This function belongs to `bas_leaf_context.c`. It is not run by any test.

`void bas_indices_in_regchain_of_all_defining_ODE (struct ba0_tableof_int_p *indices, struct bas_leaf_context *U, struct bad_regchain *C)` [Function]
Fill *indices* with the indices in C of all the y defining ODE, for all y belonging to the `smap.Y` field of U. If y has no defining ODE in C then the corresponding entry in *indices* is set to -1. This function belongs to `bas_leaf_context.c`. It is run by 19 tests, including 12 in the `bas` library and 7 in the `DifferentialAlgebra` package.

`void bas_duplicate_mumax_leaf_context (struct bas_leaf_context *U, struct ba0_tableof_int_p *mumax, ba0_int_p n)` [Function]
Append n copies of *mumax* at the end of the `mumax` field of U. This function belongs to `bas_leaf_context.c`. It is run by 1 tests, including 1 in the `bas` library and 0 in the `DifferentialAlgebra` package.

`void bas_set_all_leaves_entries_leaf_context (struct bas_leaf_context *U, struct bav_tableof_symbol *Y, struct baz_prolongation_pattern *Ybar, struct bap_tableof_polynom_mpz *ineqs, struct bav_variable *q, struct bav_symbol *x)` [Function]

Fill the fields of U which are determined by Y, *Ybar*, *ineqs*, q and x only. These fields are `smap`, `S` and `q`. The fields `mumax`, `binomials` and `point` are reset. All other tables are allocated to the size of *smap.Y* and get initialized. This function is called only once by `bas_Denef_Lipshitz`. The field `S` is assigned the list of all the polynomials of *ineqs* which are constant with respect to x. Exceptions `BAV_ERRDSY`, `BAZ_ERRIND` and `BAZ_ERRPAT` may be raised. This function belongs to `bas_leaf_context.c`. It is run by 63 tests, including 53 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_single_leaf_entries_leaf_context (struct bas_leaf_context *U, struct bad_regchain *C, struct bad_base_field *K)` [Function]

Fill the fields of U which are determined by C and K. It is assumed that `bas_set_all_leaves_entries_leaf_context` has already been called. This function is called for each regular differential chain produced by the differential elimination stage. The fields are `ode`, `order` and `sep`.

The field `kappa` is updated: each entry of `kappa` is set to 0 if it is undefined (i.e. equal to -1) and the separant of the corresponding ODE belongs to K.

The field `mumax` is assigned a first table, whose entries are filled with -1.

Exception `BA0_ERRALG` is raised if an entry of `kappa` is undefined and the separant of its associated ODE does not belong to K. This function belongs to `bas_leaf_context.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_prolongate_binomials_leaf_context (struct bas_leaf_context *U, ba0_int_p degree)` [Function]

Prolongate the `binomials` field of U up to the binomial polynomial of degree *degree*. These polynomials depend on the variable provided by the field `q` of U. The successive values of the `binomials` table are:

$$[1], \quad [1,\ q], \quad [1,\ q(q-1)/2], \quad \ldots$$

This function belongs to `bas_leaf_context.c`. It is run by 53 tests, including 43 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_printf_leaf_context (void *U0)` [Function]

The general printing function for leaf contexts. It can be called through `ba0_printf/%leaf_context`. This function belongs to `bas_leaf_context.c`. It is run by 3 tests, including 3 in the `bas` library and 0 in the `DifferentialAlgebra` package.

9.5 Tasks

Description of type `bas_task`. This data structure is a working data structure used by the `bas_Denef_Lipshitz_leaf` function. It mostly corresponds to the *task* data structure defined in [BLV25].

It is meaningful with respect to some `struct bas_leaf_context` data structure. In this documentation, some field identifiers are task or leaf context ones.

Let i be the index of some y in `smap.Y`. Let f denote the defining polynomial of y. Let s denote the separant of the defining ODE of y.

Then `sigma.tab[i]` is a monotone bound initialized from the fields `C` and `S`, extra equations and optional variables. It increases during processing. It never decreases. It is equal to -1 if y has no defining equation.

The field `mu.tab[i]` contains the prolongation order for f. Roughly speaking, this means that the field `C` contains the polynomials $f^{(j)}(y_i)$ for $0 \leq j \leq$ `mu.tab[i]`. However, this statement is not completely correct because `C` is a regular differential chain obtained by processing these polynomials by a non-differential regular chain decomposition algorithm.

The field `C` is a regular differential chain containing the ODE to be solved and the prolongation equations constraining the coefficients of their formal power series solutions.

The field `P` contains a temporary list of constant equations waiting for being processed in order to extend `C`. It is reset to the empty list each time it is processed.

The field `S` contains constant inequations to be taken into account while extending `C`. Its elements are never removed nor simplified unless they are equivalent to some nonzero base field element, in which case they are removed from S.

The field `lt_k.tab[i]` contains the last tried value for the valuation `k.tab[i]` of s. Its value is bounded by `kappa.tab[i]`. If undefined, it is equal to -1.

Then field `k.tab[i]` contains the valuation of the series obtained by evaluating s at the formal power series whose coefficients are constrained by `C` and S. It is equal to -1 if y has no defining ODE or its value has not yet been secured.

The field `fn.tab[i]` contains the Hurwitz coefficients of f with respect to the valuation stored in `k.tab[i]`. This field is undefined if `k.tab[i]` is undefined else it contains $k+1$ differential polynomials denoted (see [DL84, Lemma 2.2])

$$[f_n,\ f_{n+1},\ f_{n+2},\ \ldots,\ f_{n+k}].$$

It is empty if y has no defining ODE or `k.tab[i]` is not computed.

The field `M_row.tab[i]` is a table containing the elements of one row of the matrix M (see [BLV25, Figure 2]). Note that the elements of `M_row` are here differential polynomials, hence not prolongation ones as in [BLV25]. The table is built incrementally. Successive values of this field are

$$[f_n], \qquad [f_{n+1},\ f'_n], \qquad [f_{n+2},\ f'_{n+1},\ f''_n], \qquad \ldots$$

It is empty if y has no defining ODE or `k.tab[i]` is not computed.

The field `lt_r.tab[i]` contains the last tried value for `r.tab[i]`, which is a row index of the matrix M built from f. It is equal to -1 if y has no defining ODE or `k.tab[i]` is not computed.

The field `deg.tab[i]` is an index in `M_row.tab[i]` hence a column index in the matrix M built from f. Its value is bounded by `lt_r.tab[i]`. It is equal to -1 if y has no defining ODE or `k.tab[i]` is not computed.

The field `r.tab[i]` contains a row index in the matrix M built from f. It is the smallest row index such that the row contains a nonzero entry modulo the ideal defined by C. The index of the rightmost such nonzero entry is then given by `deg.tab[i]`. It is equal to -1 if y has no defining ODE or `k.tab[i]` is not computed.

The field `coeffs.tab[i]` contains the normal forms with respect to `C` of the order zero prolongation polynomials of the elements of `M_row.tab[i]`. It is a table of rational fractions containing `deg.tab[i]` + 1 elements. It is empty if y has no defining ODE or `r.tab[i]` is not computed.

The fields `lead_poly.tab[i]` and `A.tab[i]` contain the polynomial $A(q)$ associated to y in two different forms: `lead_poly.tab[i]` contains the normal form of the leading polynomial (as defined in [BLV25]) with respect to the regular differential chain `C`; while `A.tab[i]` contains the primitive part of the numerator of that fraction i.e. the polynomial whose roots permit to compute `gamma.tab[i]`. Their degrees in q are stored in the field `deg`. If undefined they contain 0.

The field `roots.tab[i]` contains the positive integer roots of $A(q)$, stored by increasing values.

The field `gamma.tab[i]` contains either 0 when `roots` is empty or 1 plus the maximal positive integer root of $A(q)$. If undefined, it contains -1.

The field `beta.tab[i]` contains the bound $2\,k + 2 + \gamma + r$ or -1 if undefined.

The field `omega.tab[i]` contains `true` if the bound `beta.tab[i]` is considered as correct, `false` otherwise.

The field `mumax` points to one element of the `mumax` field stored in the leaf context associated to the task.

The field `number` identifies the current task in a `struct bas_DL_tree`. This type declaration belongs to `bas_task.h`.

```
struct bas_task
{
  struct ba0_tableof_int_p sigma;

  struct ba0_tableof_int_p mu;

  struct ba0_tableof_int_p lt_k;

  struct bad_regchain C;
  struct bap_listof_polynom_mpz *P;
  struct bap_listof_polynom_mpz *S;

  struct ba0_tableof_int_p k;
  struct bap_tableof_tableof_polynom_mpz fn;
  struct bap_tableof_tableof_polynom_mpz M_row;

  struct ba0_tableof_int_p lt_r;
  struct ba0_tableof_int_p deg;
  struct ba0_tableof_int_p r;

  struct baz_tableof_tableof_ratfrac coeffs;
```

```
  struct baz_tableof_ratfrac lead_poly;
  struct baz_tableof_ratfrac A;
  struct ba0_tableof_tableof_mpz roots;
  struct ba0_tableof_int_p gamma;
  struct ba0_tableof_int_p beta;

  struct ba0_tableof_int_p omega;

  struct ba0_tableof_int_p *mumax;
  ba0_int_p number;
};
```

Description of type `bas_typeof_action_on_task`. This data type is used to determine which action needs be undertaken over a given task. This type declaration belongs to `bas_task.h`.

```
enum bas_typeof_action_on_task
{
  bas_nothing_to_do_task,
  bas_discard_task,
  bas_k_to_secure_task,
  bas_r_to_secure_task,
  bas_beta_to_compute_task,
  bas_beta_to_recompute_task
};
```

`char * bas_typeof_action_on_task_to_string (enum bas_typeof_action_on_task` *type*`)` [Function]

Return a string encoding for *type*. The encoding is given by the following table

```
"not"  bas_nothing_to_do_task
"dis"  bas_discard_task
"k"    bas_k_to_secure_task
"r"    bas_r_to_secure_task
"beta" bas_beta_to_compute_task
"bagn" bas_beta_to_recompute_task
```

This function belongs to `bas_task.c`. It is run by 3 tests, including 3 in the `bas` library and 0 in the `DifferentialAlgebra` package.

`enum bas_typeof_action_on_task bas_get_action_on_task (struct bas_task *`*T*`, struct bas_leaf_context *`*U*`)` [Function]

This function belongs to `bas_task.c`. It is run by 53 tests, including 43 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_init_task (struct bas_task *`*T*`)` [Function]

Initialize T to the empty task. This function belongs to `bas_task.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`struct bas_task * bas_new_task (void)` [Function]

Allocate a new task, initialize it and return it. This function belongs to `bas_task.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_leaf_context_task (struct bas_tableof_task *tasks, struct bas_leaf_context *U, struct bas_DL_tree *tree, struct bap_tableof_polynom_mpz *eqns, struct bad_regchain *C, struct bad_base_field *K, struct bav_tableof_variable *vars)` [Function]

Assign an initial value to the top element of *tasks* using U, *eqns*, C and K. The parameter C contains the initial regular differential chain to which the `bas_Denef_Lipshitz_leaf` function is applied. The parameter *eqns* contains further equations to be taken into account. It is allowed to be zero. The field `sigma` is initialized using the subscripted variables occurring in C, the field `S` of U, *eqns* and *vars* (if nonzero). The regular chain decomposition algorithm is applied over the prolongation equations required by `sigma`. Eventually, the top element of *tasks* is overwritten by finitely many regular differential chains whose `P` fields are empty. This function belongs to `bas_task.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_but_change_regchain_and_mumax_task (struct bas_task *dst, struct bas_task *src, struct bad_regchain *C, struct ba0_tableof_int_p *mumax)` [Function]

Assign *src* to *dst* except the `C` field of *dst* which is set to C and the `mumax` pointer which is stored in the `mumax` field of *dst*. This function belongs to `bas_task.c`. It is run by 38 tests, including 30 in the `bas` library and 8 in the `DifferentialAlgebra` package.

`void bas_set_task (struct bas_task *dst, struct bas_task *src)` [Function]

Assign *src* to *dst*. This function belongs to `bas_task.c`. It is run by 36 tests, including 29 in the `bas` library and 7 in the `DifferentialAlgebra` package.

`void bas_secure_k_task (struct bas_tableof_task *tasks, struct bas_leaf_context *U, struct bas_DL_tree *tree, struct bad_base_field *K)` [Function]

Rewrite the top element of *tasks* with finitely many tasks for which either k is defined or `lt_k` has increased. The process involves new prolongation equations and a regular chain decomposition over K. Exception `BA0_ERRALG` is raised if the field k of the top element of *tasks* is defined for each differential indeterminate. This function belongs to `bas_task.c`. It is run by 53 tests, including 43 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_secure_r_task (struct bas_tableof_task *tasks, struct bas_leaf_context *U, struct bas_DL_tree *tree, struct bad_base_field *K)` [Function]

Rewrite the top element of *tasks* with finitely many tasks for which either r and `deg` are defined or `lt_r` has increased. The field `fn` is computed. The computation of the field `M_row` is initiated. The process involves new prolongation equations and a regular chain decomposition over K. Exception `BA0_ERRALG` is raised if the field r of the top element of *tasks* is defined for each differential indeterminate. This function belongs to `bas_task.c`. It is run by 53 tests, including 43 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_compute_beta_task (struct bas_tableof_task *tasks, struct bas_leaf_context *U, struct bas_DL_tree *tree, struct bad_base_field *K)` [Function]

Rewrite the top element of *tasks* with finitely many tasks for which either a first value for *beta* is computed. The process involves new prolongation equations and a regular chain decomposition over K. The field `omega` of all new tasks is set to `false`. Exception `BA0_ERRALG` is raised if the field *beta* of the top element of *tasks* is defined for each differential indeterminate. This function belongs to `bas_task.c`. It is run by 53 tests, including 43 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_recompute_beta_task (struct bas_tableof_task *tasks, struct bas_leaf_context *U, struct bas_DL_tree *tree, struct bad_base_field *K)` [Function]

Rewrite the top element of *tasks* with finitely many tasks for which either the field `omega` is `true` or at least one coefficient of `coeffs` has been further reduced by `C`. The process involves new prolongation equations and a regular chain decomposition over K. Exception `BA0_ERRALG` is raised if the field `omega` of the top element of *tasks* is `true` for each differential indeterminate. This function belongs to `bas_task.c`. It is run by 50 tests, including 40 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_nonnegative_integer_roots (struct ba0_tableof_mpz *T, struct bap_polynom_mpz *P, struct bav_variable *v, struct bad_regchain *A)` [Function]

Assign to T the nonnegative integer roots of P, viewed as a univariate polynomial in v, with coefficients taken modulo the ideal defined by A. If v is `BAV_NOT_A_VARIABLE`, it is supposed to be the leader of P. The roots are sorted by increasing value. This function belongs to `bas_positive_integer_roots.c`. It is run by 53 tests, including 43 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_printf_task (void *T0)` [Function]

The general printing function for tasks. It can be called through `ba0_printf/%task`. This function belongs to `bas_task.c`. It is run by 3 tests, including 3 in the `bas` library and 0 in the `DifferentialAlgebra` package.

`ba0_int_p bas_Denef_Lipshitz_leaf (struct bas_tableof_pair *pairs, struct bas_DL_tree *tree, struct bas_leaf_context *U, struct bad_regchain *C, struct bad_base_field *K)` [Function]

This function is a sub-function of `bas_Denef_Lipshitz_aux`. The regular differential chain C has been produced by `bad_Rosenfeld_Groebner` over some initial differential system, over K. The leaf context U is such that the field `kappa` is defined for each differential indeterminate for which a formal power series solution is sought and a defining differential equation with a non numeric separant is present. The function appends to *pairs* finitely many pairs obtained by extending C with prolongation equations. It returns a bound for its input system, as defined in [BLV25]. This function belongs to `bas_Denef_Lipshitz.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

9.6 Hurwitz coefficients

`void bas_Hurwitz_coeffs (struct bap_tableof_polynom_mpz *fn, struct bap_polynom_mpz *P, struct bap_polynom_mpz *P_2k_2, ba0_int_p k, struct bav_symbol *x)` [Function]

Assign to *fn* the table $[f_n, f_{n+1}, \ldots, f_{n+k}]$ of the Hurwitz coefficients defined by

$$P^{(2\,k+2)} \;=\; y^{(n+2\,k+2)}\,f_n \;+\; y^{(n+2\,k+1)}\,f_{n+1} \;+\; \cdots \;+\; y^{(n+k+2)}\,f_{n+k} \;+\; f_{n+k+1}$$

where the leader of P is $y^{(n)}$, the f_i are polynomials in y of order at most i and f_n is the separant of P. Differentiations are performed with respect to x.

If nonzero, *P_2k_2* must contain $P^{(2\,k+2)}$.

The formula is due to Hurwitz (1889). It is used in [DL84, Lemma 2.2, page 215]. This function belongs to `bas_Hurwitz.c`. It is run by 56 tests, including 46 in the `bas` library and 10 in the `DifferentialAlgebra` package.

9.7 Pairs

Description of type `bas_pair`. This data structure is used to store the output of the `bas_Denef_Lipshitz` algorithm. It corresponds to *pairs* as defined in [BLV25]. Its fields are copies of the corresponding fields taken from some `struct bas_leaf_context` and some `struct bas_task` data structures.

The field `guaranteed` indicates if the pair is *guaranteed* as defined in [BLV25].

Some of the fields permit to resume the computations of the `bas_Denef_Lipshitz` algorithm: see `bas_prolongate_pair` and `bas_Denef_Lipshitz_resume`.

The other ones store data used by elementary query functions of the Python `DifferentialAlgebra` package which we do not want to recompute. This type declaration belongs to `bas_pair.h`.

```
struct bas_pair
{
  bool guaranteed;
// fields which are necessary to resume computations
  struct bav_tableof_symbol Y;
  struct baz_prolongation_pattern Ybar;
  struct bav_symbol *x;
  struct ba0_tableof_int_p kappa;
  struct bad_regchain C;
  struct bap_listof_polynom_mpz *S;
  struct bav_variable *q;
// fields for convenience only
  struct ba0_tableof_int_p order;
  struct ba0_tableof_int_p k;
  struct ba0_tableof_int_p r;
  struct baz_tableof_ratfrac lead_poly;
  struct baz_tableof_ratfrac A;
//  struct baz_tableof_ratfrac det_A;
  struct ba0_tableof_int_p gamma;
  struct ba0_tableof_int_p mu;
  struct ba0_tableof_int_p sigma;
  struct ba0_tableof_int_p beta;
};
```

`void bas_init_pair (struct bas_pair *pair)` [Function]
Initialize *pair* to the empty pair. This function belongs to `bas_pair.c`. It is run by 57 tests, including 47 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`struct bas_pair * bas_new_pair (void)` [Function]
Allocate a new pair, initialize it and return it. This function belongs to `bas_pair.c`. It is run by 51 tests, including 41 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_pair (struct bas_pair *dst, struct bas_pair *src)` [Function]
Assign *src* to *dst*. This function belongs to `bas_pair.c`. It is run by 13 tests, including 3 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_context_task_pair (struct bas_pair *pair, bool guaranteed, struct bas_leaf_context *U, struct bas_task *Z)` [Function]
Assign to *pair* the pair defined by *guaranteed*, U and Z. The result is obtained by copying the corresponding fields of U and Z, except the field `S` of *pair*, which is the union of the fields `S` of U and Z. This function belongs to `bas_pair.c`. It is run by 50 tests, including 40 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_constant_variables_pair (struct ba0_tableof_int_p *sigma, struct bav_tableof_variable *omega, struct bas_pair *pair)` [Function]
Assign to *sigma* the maximum of the subscripts of the variables `y[i]` occurring in the field `C` of *pair*, where `y[i]` denotes some subscripted variable associated to some element of the field `Y` of *pair* (-1 if undefined). The ith entry of *sigma* corresponds to the ith entry of the field `Y` of *pair*. Assign to *omega* all the constant variables occurring in the field `C` of *pair* which are not the leaders of any element of `C`. This function belongs to `bas_pair.c`. It is run by 5 tests, including 4 in the `bas` library and 1 in the `DifferentialAlgebra` package.

`void bas_series_coefficients_pair (struct baz_tableof_tableof_ratfrac *T, struct bas_pair *pair)` [Function]
Assign to T the normal forms of the coefficients of the series defined by *pair*. The ith entry of T corresponds to the the ith entry `y` of the field `Y` of *pair*. It contains the normal forms of `y[0]`, `y[1]`, `y[2]/2`, `...`, `y[n]/n!` where n denotes the maximum of the subscripts of the `y[i]`, occurring in the field `C` of *pair*. This function belongs to `bas_pair.c`. It is run by 3 tests, including 2 in the `bas` library and 1 in the `DifferentialAlgebra` package.

`void bas_select_pairs (struct bas_tableof_pair *dst, struct bas_tableof_pair *src, bool guaranteed)` [Function]
Assign to *dst* the table of the elements of *src* whose `guaranteed` field is equal to *guaranteed*. This function belongs to `bas_pair.c`. It is run by 39 tests, including 39 in the `bas` library and 0 in the `DifferentialAlgebra` package.

`unsigned ba0_int_p bas_sizeof_pair (struct bas_pair *pair, enum ba0_garbage_code code)` [Function]

Return the size of the memory needed to perform a copy of *pair*. If *code* is `ba0_embedded` then *pair* is supposed to be embedded in a larger data structure so that its size is not taken into account. This function belongs to `bas_pair.c`. It is run by 11 tests, including 1 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_switch_ring_pair (struct bas_pair *pair, struct bav_differential_ring *R)` [Function]

This low level function should be used in conjunction with `bav_set_differential_ring`: if R is a ring obtained by application of `bav_set_differential_ring` to the ring *pair* refers to, then this function makes *pair* refer to R. This function belongs to `bas_pair.c`. It is run by 11 tests, including 1 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void * bas_scanf_pair (void *pair0)` [Function]

The general parsing function for pairs. It can be called through `ba0_scanf/%pair`. This function belongs to `bas_pair.c`. It is run by 47 tests, including 47 in the `bas` library and 0 in the `DifferentialAlgebra` package.

`void bas_printf_pair (void *pair0)` [Function]

The general printing function for pairs. It can be called through `ba0_printf/%pair`. This function belongs to `bas_pair.c`. It is run by 41 tests, including 41 in the `bas` library and 0 in the `DifferentialAlgebra` package.

`ba0_int_p bas_Denef_Lipshitz_aux (struct bas_tableof_pair *pairs, struct bas_DL_tree *DL_tree, ba0_int_p beta_in, struct bas_leaf_context *U, struct bad_intersectof_regchain *ideal, struct bad_splitting_tree *RG_tree, ba0_int_p number, struct bad_base_field *K)` [Function]

The ideal *ideal* and the splitting tree *RG_tree* have been produced by the `bad_Rosenfeld_Groebner` algorithm over some initial differential system, possibly with *ad hoc* equations and over K. The argument *number* is the number of some node in *RG_tree*. The leaf_context U has been partially initialized: the data related to the differential indeterminates for which a formal power series solution is sought have been computed but not the data related to the defining ODE of these differential indeterminates. The function appends to *pairs* finitely many pairs obtained by processing the components of *ideal* which belong to the splitting tree starting at vertex *number*. It assigns to *beta* the bound defined by all these pairs and the differential reductions performed by the differential elimination process. This function is a sub-function of `bas_Denef_Lipshitz`. This function belongs to `bas_Denef_Lipshitz.c`. It is run by 49 tests, including 39 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_Denef_Lipshitz (struct bas_tableof_pair *pairs, struct bas_DL_tree *tree, struct bap_tableof_polynom_mpz *eqns, struct bap_tableof_polynom_mpz *ineqs, struct ba0_tableof_string *properties, struct bav_tableof_symbol *Y, struct baz_prolongation_pattern *Ybar, struct bav_variable *q, struct bav_symbol *x)` [Function]

The tables *eqns* and *ineqs* contain the differential system of equations and inequations to be processed. The table *properties* provides the properties needed by the `bad_Rosenfeld_Groebner` function. The table Y contains the differential indeterminates for which formal power series solutions are sought. The prolongation pattern *Ybar* associates to each differential indeterminate in Y the subscripted variables used to denote the coefficients of its formal power series solution. The variable q contains the variable used for the polynomials $A(q)$ (see `struct bas_task`). The symbol x provides the derivation of the problem. The function applies an experimental version of the `DenefLipshitz` algorithm over all these input data, using the strategy given by *beta_control* and store the result in *pairs*. This function belongs to `bas_Denef_Lipshitz.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_Denef_Lipshitz_resume (struct bas_tableof_pair *pairs, struct bas_pair *pairs0, struct bap_tableof_polynom_mpz *eqns, struct bap_tableof_polynom_mpz *ineqs)` [Function]

The pair *pairs0* has been produced by the `bas_Denef_Lipshitz_leaf` or the `bas_Denef_Lipshitz_resume` function. Resume the computation, taking into account the new equations and inequations stored in *eqns* and *ineqs*. Append the new pairs to *pairs*. This function belongs to `bas_Denef_Lipshitz.c`. It is run by 9 tests, including 8 in the `bas` library and 1 in the `DifferentialAlgebra` package.

`void bas_prolongate_pair (struct bas_pair *dst, struct bas_pair *src, struct bav_tableof_variable *vars)` [Function]

Assign to *dst* the pair obtained by prolongating *src* using *vars*. The table *vars* is supposed to contained subscripted variables. These variables provide the prolongation limits. This function belongs to `bas_Denef_Lipshitz.c`. It is run by 3 tests, including 2 in the `bas` library and 1 in the `DifferentialAlgebra` package.

9.8 DL Trees

9.8.1 Edges

Description of type `bas_typeof_DL_edge`. This data type is a subtype of `bas_DL_edge`. It permits to associate a type to an edge in the DL tree generated by the `DenefLipshitz_leaf` algorithm. This type declaration belongs to `bas_DL_edge.h`.

```
enum bas_typeof_DL_edge
{
// not an edge
  bas_none_edge,
// investigation of some coefficient being zero
  bas_vanishing_edge,
```

```
// investigation of some coefficient being nonzero
  bas_non_vanishing_edge,
// splitting due to regular chain decomposition
  bas_RG_edge
};
```

Description of type `bas_DL_edge`. This data type is a subtype of `struct bas_DL_vertex` and `struct bas_DL_tree`. The `bas_DenefLipshitz_leaf` function is led to consider some polynomials which may possibly vanish. In such cases, two cases are generated: the one in which the polynomial under consideration vanish and the one in which it does not. The two corresponding types are `bas_vanishing_edge` and `bas_non_vanishing_edge`. The `bas_DenefLipshitz_leaf` function is also led to generate prolongation equations, which are simplified by a regular chain decomposition algorithm, possibly yielding some case discussion. The corresponding branches have type `bas_RG_edge`. This type declaration belongs to `bas_DL_edge.h`.

```
struct bas_DL_edge
{
// the type of the edge
  enum bas_typeof_DL_edge type;
// the number of the source vertex
  ba0_int_p src;
// the number of the target vertex
  ba0_int_p dst;
};
```

`void bas_init_DL_edge (struct bas_DL_edge *E)` [Function]

Initialize *E*. This function belongs to `bas_DL_edge.c`. It is run by 19 tests, including 11 in the `bas` library and 8 in the `DifferentialAlgebra` package.

`struct bas_DL_edge * bas_new_DL_edge (void)` [Function]

Allocate a new edge, initialize it and return it. This function belongs to `bas_DL_edge.c`. It is run by 19 tests, including 11 in the `bas` library and 8 in the `DifferentialAlgebra` package.

`char * bas_typeof_DL_edge_to_string (enum bas_typeof_DL_edge type)` [Function]

Return a string encoding for *type*. The encoding is given by the following table

```
bas_none_edge
"van" bas_vanishing_edge
"nvn" bas_non_vanishing_edge
"RG"  bas_RG_edge
```

This function belongs to `bas_DL_edge.c`. It is not run by any test.

`void bas_set_DL_edge (struct bas_DL_edge *E, struct bas_DL_edge *F)` [Function]

Assign *F* to *E*. This function belongs to `bas_DL_edge.c`. It is not run by any test.

`void bas_set_tsd_DL_edge (struct bas_DL_edge *E, enum bas_typeof_DL_edge type, ba0_int_p src, ba0_int_p dst)` [Function]

Assign *src*, *dst* and *type* to the corresponding fields of *E*. This function belongs to `bas_DL_edge.c`. It is run by 19 tests, including 11 in the `bas` library and 8 in the `DifferentialAlgebra` package.

`void bas_printf_DL_edge (void *A)` [Function]

The printing function for DL edges. It is called by `ba0_printf/%DL_edge`. This function belongs to `bas_DL_edge.c`. It is not run by any test.

9.8.2 Vertices

Description of type `bas_typeof_consistency_vertex`. This data type is a subtype of `bas_DL_vertex`. It determines the consistency information of one vertex of a DL tree. This type declaration belongs to `bas_DL_vertex.h`.

```
enum bas_typeof_consistency_vertex
{
// the vertex is inconsistent
  bas_inconsistent_vertex,
// the vertex is rejected because kappa is exceeded
  bas_rejected_vertex,
// the consistency of the vertex is uncertain (at least when created)
  bas_uncertain_vertex,
// the vertex is consistent
  bas_consistent_vertex
};
```

Description of type `bas_DL_vertex`. This data type is a subtype of `bas_DL_tree`. It permits to describe one vertex of the tree. Each vertex corresponds to a task Z, which is identified by its *number*.

The field `number` contains the number of the vertex.

The field `consistency` provides the consistency information for the vertex. The default value is `bas_uncertain_vertex`.

The field `edges` contains the table of the edges starting from the vertex towards other vertices of the DL tree.

The field `action` contains the action to be undertaken on the vertex. The field `y` contains the differential indeterminate with respect to which the action is undertaken. The fields `k`, `r` and `deg` contain the entries of the corresponding fields of Z which apply to `y`. This type declaration belongs to `bas_DL_vertex.h`.

```
struct bas_DL_vertex
{
// the number of the vertex which is also the number of the quadruple
  ba0_int_p number;
// the consistency information for the vertex
  enum bas_typeof_consistency_vertex consistency;
// the edges towards the children of the vertex
  struct bas_tableof_DL_edge edges;
// information of the process undertaken by the vertex
  enum bas_typeof_action_on_task action;
  struct bav_symbol *y;
  ba0_int_p k;
  ba0_int_p r;
  ba0_int_p deg;
};
```

`void bas_init_DL_vertex (struct bas_DL_vertex *V)` [Function]

Initialize V to an empty vertex with number `BAS_NOT_A_NUMBER`. This function belongs to `bas_DL_vertex.c`. It is run by 26 tests, including 16 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`struct bas_DL_vertex * bas_new_DL_vertex (void)` [Function]
Allocate a new vertex, initialize it and return it. This function belongs to `bas_DL_vertex.c`. It is run by 26 tests, including 16 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_reset_DL_vertex (struct bas_DL_vertex *V, ba0_int_p number)` [Function]
Reset V to an empty vertex but set its number to *number*. This function belongs to `bas_DL_vertex.c`. It is run by 26 tests, including 16 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_DL_vertex (struct bas_DL_vertex *dst, struct bas_DL_vertex *src)` [Function]
Assign *src* to *dst*. This function belongs to `bas_DL_vertex.c`. It is not run by any test.

`void bas_set_aykrd_DL_vertex (struct bas_DL_vertex *V, enum bas_typeof_action_on_task action, struct bav_symbol *y, ba0_int_p k, ba0_int_p r, ba0_int_p deg)` [Function]
Set the fields `action`, `y`, `k`, `r` and `deg` of V to *action*, y, k, r and *deg*. This function belongs to `bas_DL_vertex.c`. It is run by 25 tests, including 15 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_printf_DL_vertex (void *A)` [Function]
The general printing function for vertices. It can be called through `ba0_printf/%DL_vertex`. This function belongs to `bas_DL_vertex.c`. It is not run by any test.

9.8.3 Trees

Description of type `bas_activity_level_DL_tree`. This data type is a subtype of `bas_DL_tree`. This type declaration belongs to `bas_DL_tree.h`.

```
enum bas_activity_level_DL_tree
{
// the DL tree is not generated
  bas_inactive_DL_tree,
// the DL tree is generated
  bas_quiet_DL_tree,
// the DL tree is generated and messages are printed
  bas_verbose_DL_tree
};
```

Description of type `bas_DL_tree`. This data type permits to describe the DL tree generated by the `bas_Denef_Lipshitz_leaf`Â function.

The field `roots` contains the numbers of the different roots of the different trees (the data structure actually contains a forrest rather than a single tree).

The field `vertices` contains the table of the tree vertices. Each vertex is associated to one task, identified by its *number*. This *number* is moreover equal to the index of the vertex in the table.

The field `number` contains the next free vertex number. It may be greater than the `alloc` field of the `vertices` table.

The field `activity` provides the level of activity of the tree. It may have three values:

- `bas_inactive_DL_tree` then the only active field if `number` and the two tables remain empty˜;
- `bas_quiet_DL_tree` then the DL tree is generated
- `bas_verbose_DL_tree` same as above but the the `bas_Denef_Lipshitz_leaf`Â function may print some further data on the standard output.

This type declaration belongs to `bas_DL_tree.h`.

```
struct bas_DL_tree
{
  enum bas_activity_level_DL_tree activity;
// the table of the roots
  struct ba0_tableof_int_p roots;
// the table of vertices
  struct bas_tableof_DL_vertex vertices;
// the next free vertex number
  ba0_int_p number;
};
```

`void bas_init_DL_tree (struct bas_DL_tree *tree)` [Function]

Initialize *tree* to an empty inactive tree. This function belongs to `bas_DL_tree.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`struct bas_DL_tree * bas_new_DL_tree (void)` [Function]

Allocate a new tree, initialize it and return it. This function belongs to `bas_DL_tree.c`. It is not run by any test.

`void bas_reset_DL_tree (struct bas_DL_tree *tree, enum bas_activity_level_DL_tree level)` [Function]

Reset *tree* to an empty tree with activity level *level*. This function belongs to `bas_DL_tree.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_DL_tree (struct bas_DL_tree *dst, struct bas_DL_tree *src)` [Function]

Assign *src* to *dst*. This function belongs to `bas_DL_tree.c`. It is not run by any test.

`void bas_add_root_DL_tree (struct bas_DL_tree *tree, ba0_int_p root)` [Function]

Store the vertex number *root* in the `roots` field of *tree*. This function belongs to `bas_DL_tree.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`ba0_int_p bas_next_number_DL_tree (struct bas_DL_tree *tree)` [Function]

Return the next available task number available in *tree*. This function does not allocate any memory. This function belongs to `bas_DL_tree.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`struct bas_DL_vertex * bas_ith_vertex_DL_tree (struct bas_DL_tree *tree, ba0_int_p number)` [Function]

Return the vertex dedicated to the task identified by *number* in *tree*. This function may perform memory allocation. This function belongs to `bas_DL_tree.c`. It is run by 26 tests, including 16 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_vertex_consistency_DL_tree (struct bas_DL_tree *tree, ba0_int_p number, enum bas_typeof_consistency_vertex consistency)` [Function]

Set the field `consistency` of the vertex *number* of *tree* to *consistency*. This function belongs to `bas_DL_tree.c`. It is run by 59 tests, including 49 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_set_aykrd_vertex_DL_tree (struct bas_DL_tree *tree, ba0_int_p number, enum bas_typeof_action_on_task action, struct bav_symbol *y, ba0_int_p k, ba0_int_p r, ba0_int_p deg)` [Function]

Set the fields `action`, `y`, `k`, `r` and `deg` of the vertex with number *number* of *tree* to *action*, *y*, *k*, *r* and *deg*. This function belongs to `bas_DL_tree.c`. It is run by 53 tests, including 43 in the `bas` library and 10 in the `DifferentialAlgebra` package.

`void bas_add_edge_DL_tree (struct bas_DL_tree *tree, enum bas_typeof_DL_edge type, ba0_int_p src, ba0_int_p dst)` [Function]

Add the edge defined by *type*, *src* and *dst* to *tree*. In the case of an inactive tree, the edge is not recorded. This function belongs to `bas_DL_tree.c`. It is run by 38 tests, including 30 in the `bas` library and 8 in the `DifferentialAlgebra` package.

`void bas_dot_DL_tree (struct bas_DL_tree *tree)` [Function]

Print *tree* as a directed graph, following the syntax of `graphviz/dot`. Edges are labelled using the encoding given by `bas_typeof_DL_edge_to_string`. The shapes of the vertices are functions of their fields `consistency` and `argument`. The correspondence is as follows:

```
ellipse       bas_uncertain_vertex
septagon      bas_rejected_vertex (kappa exceeded)
box           bas_consistent_vertex
pentagon      bas_inconsistent_vertex
```

This function belongs to `bas_DL_tree.c`. It is not run by any test.

`void bas_printf_DL_tree (void *A)` [Function]

The printing function for DL trees. It is called by `ba0_printf/%DL_tree`. This function belongs to `bas_DL_tree.c`. It is not run by any test.

10 The bai library

10.1 Introduction

The `bai` library implements tools for connecting the regular differential chains produced by the `bad` library to numerical integrators of differential equations. In particular, it contains a C adaptation of the Fortran DAE solver `radau5` first implemented by E. Hairer [Hairer] and described in [HW96]. Though the numerical algorithm is the one of `radau5`, the C code is not a straightforward translation of the Fortran code: it has required quite some refactoring.

- [HW96] Ernst Hairer and Gerhard Wanner. *Solving ordinary differential equations II. Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics. vol. 14, 2nd edition. 1996
- [Hairer] `https://www.unige.ch/~hairer/software.html`

10.1.1 Basic types, constants and naming conventions

The name of every exported object defined in the library starts by `bai_`. Functions starting by `bai_init` initialize raw data structures (constructors). Functions starting by `bai_new` dynamically allocate data structures and apply the corresponding `bai_init` functions on them. Functions starting by `bai_reset` reset data structures which are assumed to be already initialized. Functions starting by `bai_set` perform assignments and systematically copy the sources to the destinations.

A *sequence of calls* to the `bai` library starts with a call to `bai_restart` and finishes by a call to `bai_terminate`.

`void bai_restart (ba0_int_p` *`time_limit`*`, ba0_int_p` *`memory_limit`*`)` [Function]

Call the `bas_restart` function with the same parameters This function belongs to `bai_common.c`. It is run by 34 tests, including 34 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_terminate (enum ba0_restart_level` *`level`*`)` [Function]

Call the `bas_terminate` function with the same parameter. This function belongs to `bai_common.c`. It is run by 34 tests, including 34 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.2 Variable bindings

Description of type `bai_external_C_function`. This declaration corresponds to the signatures of the functions associated to external variables. See `enum bai_typeof_variable`. This type declaration belongs to `bai_variable_bindings.h`.

```
typedef double bai_external_C_function (
    struct bai_variable_bindings *,
    double,
    double *);
```

Description of type `bai_typeof_variable`. This data type permits to assign a type to the derivatives occurring in a polynomial differential system.

- `bai_constant_variable`: the derivative is a constant. It will be associated to some floating point number. This type corresponds as well to a plain constant `a`, `b`, \dots as to an initial value `y[0]`, `z[1]`, \dots
- `bai_independent_variable`: the variable is the independent variable.
- `bai_computed_variable`: the derivative is a derivative of some differential indeterminate whose values are produced by a numerical integrator.
- `bai_external_variable`: the derivative denotes a function. It will be associated to some function of type `bai_external_C_function`.

This type declaration belongs to `bai_variable_bindings.h`.

```
enum bai_typeof_variable
{
  bai_constant_variable,
  bai_independent_variable,
  bai_computed_variable,
  bai_external_variable
};
```

Description of type `bai_variable_bindings`. The `struct bai_variable_bindings` data structure aims at mapping variables to constants, computed variables and external variables.

The field `smap` is a data structure which permits to associate derivatives of computed variables such as `y[x]`, `z[x,x]` to the corresponding series coefficients `y[1]`, `z[2]`.

The fields `vars`, `types` and `index` have the same sizes. The field `vars` contains all the derivatives added to the data structure. The field `types` contains the type associated to each of these derivatives.

If an entry of `vars` is a constant variable then the corresponding `index` entry contains an index in both `c` and `c_vars`. If an entry of `vars` is an external variable then the corresponding `index` entry contains an index in both `u` and `u_vars`. If an entry of `vars` is the independent variable then the corresponding `index` entry contains `BAO_NOT_AN_INDEX`.

Last, if an entry of `vars` is a computed variable then the corresponding `index` entry contains either `BAO_NOT_AN_INDEX` or an index in both `y` and `y_vars`. The computed variables whose `index` entries are `BAO_NOT_AN_INDEX` correspond to derivatives whose defining ODE can be solved for and thus derivatives whose values do not need to be stored in `y`. They correspond to leaders of differential polynomials, with positive orders, degree one and constant coefficient. They are called *extra* computed variables.

The fields `c_vars` and `c` have the same sizes. The field `c_vars` contains all the derivatives occurring in `vars` which are constant variables. The field `c` contains their values.

The field `x` contains the value associated to the independent variable.

The fields `y_vars` and `y` have the same sizes. The field `y_vars` contains all the derivatives occurring in `vars` which are computed variables, but not *extra*. The field `y` contains their values.

The fields `u_vars` and `u` have the same sizes. The field `u_vars` contains all the derivatives occurring in `vars` which are external variables. The field `u` contains pointers to their corresponding external functions. External functions are not yet implemented. This type declaration belongs to `bai_variable_bindings.h`.

```
struct bai_variable_bindings
```

```
{
  struct baz_series_mapping smap;
  struct bav_tableof_variable vars;
  struct bai_tableof_typeof_variable types;
  struct ba0_tableof_int_p index;
  struct bav_dictionary_variable dict;
  struct bav_tableof_variable c_vars;
  struct ba0_tableof_double c;
  double x;
  struct bav_tableof_variable y_vars;
  struct ba0_tableof_double y;
  struct bav_tableof_variable u_vars;
  struct bai_tableof_external_function u;
};
```

`void bai_add_computed_variable_bindings (struct bai_variable_bindings *B, struct bav_variable *v)` [Function]

If v is not already present in B, add v to the computed variables of B. In particular entries are allocated for v in the fields `y` and `y_vars` of B. Exception `BA0_ERRALG` is raised if v is already present in B with a different type. This function belongs to `bai_variable_bindings.c`. It is run by 34 tests, including 34 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_add_constant_variable_bindings (struct bai_variable_bindings *B, struct bav_variable *v, double d)` [Function]

If v is not already present in B, add v to the constant variables of B with value d. If v is already present, change its value to d. Exception `BA0_ERRALG` is raised if v is already present in B with a different type. This function belongs to `bai_variable_bindings.c`. It is run by 24 tests, including 24 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_add_extra_computed_variable_bindings (struct bai_variable_bindings *B, struct bav_variable *v)` [Function]

If v is not already present in B, add v to B with type `bai_computed_variable` and index `BA0_NOT_AN_INDEX`. No entries are allocated for v in the fields `y` and `y_vars` of B. Exception `BA0_ERRALG` is raised if v is already present in B with a different type. This function belongs to `bai_variable_bindings.c`. It is run by 8 tests, including 8 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_add_point_constant_variable_bindings (struct bai_variable_bindings *B, struct ba0_point_double *csts)` [Function]

Add each variable occurring in *csts* to B as a constant, using `bai_add_constant_variable_bindings`. This function belongs to `bai_variable_bindings.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_add_regchain_computed_variable_bindings (struct bai_variable_bindings *B, struct bad_regchain *C)` [Function]

Add to B all computed variables needed for integrating C. These are all the derivatives occurring in C. Derivatives v which have positive orders, are leaders of some $f \in C$ with degree 1, constant coefficients and which do not occur in any other element of C

are recorded as *extra* computed variables. If v is added to B and v is a derivative of w then w is added to B also. This function belongs to `bai_variable_bindings.c`. It is run by 29 tests, including 29 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_add_regchain_constant_variable_bindings (struct bai_variable_bindings *B, struct bad_regchain *C)` [Function]

Add all constant variables occurring in C to B. Constant variables whose values are uniquely defined by some elements of C are assigned the corresponding values. The other constant variables are assigned `nan`. This function belongs to `bai_variable_bindings.c`. It is run by 25 tests, including 25 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_add_regchain_variable_bindings (struct bai_variable_bindings *B, struct bad_regchain *C)` [Function]

Add to B all variables occurring in C. See `bai_add_regchain_constant_variable_bindings` and `bai_add_regchain_computed_variable_bindings`. This function belongs to `bai_variable_bindings.c`. It is run by 25 tests, including 25 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_add_spx_variable_bindings (struct bai_variable_bindings *B, struct bav_tableof_symbol *Y, struct baz_prolongation_pattern *Ybar, struct bav_symbol *x)` [Function]

Fill the field `smap` of B using Y, $Ybar$ and x. This function belongs to `bai_variable_bindings.c`. It is run by 34 tests, including 34 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_init_variable_bindings (struct bai_variable_bindings *B)` [Function]

Initialize the mapping B. This function belongs to `bai_variable_bindings.c`. It is run by 34 tests, including 34 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`struct bai_variable_bindings * bai_new_variable_bindings (void)` [Function]

Allocate a new mapping, initialize it and return it. This function belongs to `bai_variable_bindings.c`. It is not run by any test.

`void bai_reset_initial_values_variable_bindings (struct bai_variable_bindings *B, struct bav_tableof_variable *vars)` [Function]

Reset the current values of computed variables from their associated initial values. Computed variables which do not have prescribed initial values are assigned `nan`. If *vars* is nonzero, it receives the table of these variables without any prescribed initial values. This function belongs to `bai_variable_bindings.c`. It is run by 16 tests, including 16 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_variable_bindings (struct bai_variable_bindings *dst, struct bai_variable_bindings *src)` [Function]

Assign *src* to *dst*. This function belongs to `bai_variable_bindings.c`. It is not run by any test.

10.3 Left hand side, mass and jacobian matrices

`void bai_mass_f_regchain_polynom_mpz (struct bap_matrixof_polynom_mpz *M, struct bap_tableof_polynom_mpz *F, struct bad_regchain *C, struct bai_variable_bindings *B)` [Function]

Assign to M and F the mass matrix and the right hand side of the semi-explicit form system obtained from C. The variables occurring in C must occur in B. This function belongs to `bai_mass_f_jacobian.c`. It is run by 25 tests, including 25 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_jacobian_f_polynom_mpz (struct bap_matrixof_polynom_mpz *J, struct bap_tableof_polynom_mpz *F, struct bai_variable_bindings *B)` [Function]

Assign to J the Jacobian matrix of the vector F (see `bai_mass_f_regchain_polynom_mpz`). This function belongs to `bai_mass_f_jacobian.c`. It is run by 18 tests, including 18 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.4 Recursive polynomials

Description of type `bai_coefficient`. This data type is a subtype of `struct bai_recursive_polynomial`. It permits to represent one coefficient of a recursive polynomial. If `type` is `bai_double_coefficient` then the coefficient is a `double` whose value is stored in the field `double_coeff`. If `type` is `bai_polynom_coefficient` then the coefficient is itself a recursive polynomial, whose value is stored in the field `polynom_coeff`. This type declaration belongs to `bai_recursive_polynomial.h`.

```
struct bai_coefficient
{
  enum bai_typeof_coefficient type;
  double double_coeff;
  struct bai_recursive_polynomial *polynom_coeff;
};
```

`void bai_set_double_coefficient (struct bai_coefficient *coeff, double d)` [Function]

Assign d to *coeff*. This function belongs to `bai_recursive_polynomial.c`. It is run by 14 tests, including 14 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_polynom_coefficient (struct bai_coefficient *coeff, struct bai_recursive_polynomial *poly)` [Function]

Assign *poly* to *coeffs* This function belongs to `bai_recursive_polynomial.c`. It is run by 12 tests, including 12 in the `bai` library and 0 in the `DifferentialAlgebra` package.

Description of type `bai_recursive_polynomial`. This data type permits to represent a recursive polynomial. The field `leader` is equal to `BAV_NOT_A_VARIABLE` if the polynomial is a numerical constant otherwise it contains its leader. The fields `type` and `index` are then copies of the corresponding entries taken from an embedding `struct bai_variable_bindings` data structure. The field `coeffs` contains the table of the polynomial coefficients. It has size zero in the case of the zero polynomial. This type declaration belongs to `bai_recursive_polynomial.h`.

```
struct bai_recursive_polynomial
{
  struct bav_variable *leader;
  enum bai_typeof_variable type;
  ba0_int_p index;
  struct bai_tableof_coefficient coeffs;
};
```

`double bai_eval_recursive_polynomial (struct bai_recursive_polynomial *poly, struct bai_variable_bindings *B, double x, double *y)` [Function]

Evaluate *poly* using the symbolic mapping given by B and the floating point values provided by x and y. Constants are evaluated using the floating point values stored in B. The argument y may be the zero pointer. In that case, the polynomial must be a constant (case of the evaluation of the mass matrix). This function belongs to `bai_recursive_polynomial.c`. It is run by 15 tests, including 15 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_init_recursive_polynomial (struct bai_recursive_polynomial *poly)` [Function]

Initialize *poly* to the zero polynomial. This function belongs to `bai_recursive_polynomial.c`. It is run by 15 tests, including 15 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`struct bai_recursive_polynomial * bai_new_recursive_polynomial (void)` [Function]

Allocate a new polynomial, initialize it and return it. This function belongs to `bai_recursive_polynomial.c`. It is run by 13 tests, including 13 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_matrixof_polynom_mpz_matrixof_recursive_polynomial (struct bai_matrixof_recursive_polynomial *dst, struct bap_matrixof_polynom_mpz *src, struct bai_variable_bindings *B)` [Function]

Assign *src* to *dst*. The variables occurring in *src* must occur in B. This function belongs to `bai_recursive_polynomial.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_polynom_mpz_recursive_polynomial (struct bai_recursive_polynomial *poly, struct bap_polynom_mpz *p, struct bai_variable_bindings *B)` [Function]

Assign p to *poly*. The variables occurring in p must occur in B. This function belongs to `bai_recursive_polynomial.c`. It is run by 15 tests, including 15 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_tableof_polynom_mpz_tableof_recursive_polynomial (struct bai_tableof_recursive_polynomial *dst, struct bap_tableof_polynom_mpz *src, struct bai_variable_bindings *B)` [Function]

Assign *src* to *dst*. The variables occurring in *src* must occur in B. This function belongs to `bai_recursive_polynomial.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_variable_recursive_polynomial (struct bai_recursive_polynomial *poly, struct bav_variable *v, struct bai_variable_bindings *B)` [Function]

Assign the polynomial v to *poly*. The variable v must occur in B. This function belongs to `bai_recursive_polynomial.c`. It is not run by any test.

Description of type `bai_radau5_recursive_polynomial_data`. The RADAU5 numerical integrator was not designed to work with BLAD. This data structure bundles in a single structure all the information needed to evaluate the system right hand side, its jacobian and mass matrices in recursive polynomial form. This type declaration belongs to `bai_radau5_recursive_polynomial_data.h`.

```
struct bai_radau5_recursive_polynomial_data
{
  struct bai_variable_bindings *B;
  struct bai_tableof_recursive_polynomial *fcn;
  struct bai_matrixof_recursive_polynomial *jac;
  struct bai_matrixof_recursive_polynomial *mas;
};
```

`void bai_init_radau5_recursive_polynomial_data (struct bai_radau5_recursive_polynomial_data *R)` [Function]

Initialize R to the empty structure. This function belongs to `bai_radau5_recursive_polynomial_data.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_regchain_radau5_recursive_polynomial_data (struct bai_radau5_recursive_polynomial_data *R, struct bad_regchain *C, struct bai_variable_bindings *B)` [Function]

Assign to R the structure obtained from B and C. This function belongs to `bai_radau5_recursive_polynomial_data.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.5 The RADAU5 DAE integrator

10.5.1 Initial values

Description of type `bai_radau5_initial_values`. This type declaration belongs to `bai_radau5_input.h`.

```
struct bai_radau5_initial_values
{
  ba0_int_p n;
  double x0;
  double xend;
  double *y0;
  double h0;
};
```

`void bai_set_nxyh_radau5_initial_values (struct bai_radau5_initial_values *iv, ba0_int_p n, double x0, double xend, double *y0, double h0)` [Function]

Set *iv* to the initial values defined by *n* (the system dimension), *x0*, *xend* (the integration interval), *y0* (the vector of initial values) and *h0* (the initial step size). If *h0* is zero then it is set to 10^{-6}. This function belongs to `bai_radau5_input.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.5.2 Tolerances

Description of type `bai_radau5_typeof_tolerance`. This type declaration belongs to `bai_radau5_tolerances.h`.

```
enum bai_radau5_typeof_tolerance
{
  bai_scalar_tolerance = 0,
  bai_vector_tolerance = 1
};
```

Description of type `bai_radau5_tolerances`. This type declaration belongs to `bai_radau5_tolerances.h`.

```
struct bai_radau5_tolerances
{
  double *rtol;
  double *atol;
  enum bai_radau5_typeof_tolerance itol;
};
```

`void bai_init_radau5_tolerances (struct bai_radau5_tolerances *tol)` [Function]

Initialize *tol* to scalar tolerances equal to 10^{-6}. This function belongs to `bai_radau5_tolerances.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_double_radau5_tolerances (struct bai_radau5_tolerances *tol, double rtol, double atol)` [Function]

Set *tol* to the scalar tolerances given by *rtol* and *atol*. This function belongs to `bai_radau5_tolerances.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_radau5_tolerances (struct bai_radau5_tolerances *dst, struct bai_radau5_tolerances *src)` [Function]

Assign *src* to *dst*. This function belongs to `bai_radau5_tolerances.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.5.3 Input

Description of type `bai_radau5_input`. This data structure contains input data for the numerical integrator The subfield `rpd` contains information which permit to connect the integrator to a symbolic blad environment. In particular, the subfield `rpd.B` stores the floating point arrays computed by the integrator (they are thus modified during the integration process). This type declaration belongs to `bai_radau5_input.h`.

```
struct bai_radau5_input
{
  struct bai_radau5_initial_values initial;
  struct bai_radau5_tolerances tolerances;
  struct bai_radau5_callbacks callbacks;
  struct bai_radau5_jacobian_options jacobian;
  struct bai_radau5_mass_options mass;
  ba0_int_p iwork[11];
  double work[10];
//  struct bai_radau5_user_data user_data;
  struct bai_radau5_recursive_polynomial_data rpd;
};
```

`void bai_init_radau5_input (struct bai_radau5_input *input)` [Function]

Initialize *input* to the empty input with some default values. This function belongs to `bai_radau5_input.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_regchain_radau5_input (struct bai_radau5_input *input, struct bad_regchain *C, struct bai_variable_bindings *B)` [Function]

Set *input* to the input data structure defined by C and B. This function belongs to `bai_radau5_input.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_input_result_radau5_integration_state (struct bai_radau5_integration_state *R, struct bai_radau5_input *input, struct bai_radau5_result *result)` [Function]

Assign starting values to R fields using *input* and *result*. This function belongs to `bai_radau5_integration_state.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.5.4 Result

Description of type `bai_radau5_result`. This data type permits to store the numerical values computed by the numerical integrator. Note that the fields `x` and `y` point to storage areas located in some other structure. This type declaration belongs to `bai_radau5_result.h`.

```
struct bai_radau5_result
{
```

```
  double *x;
  double h;
  double *y;
  enum bai_radau5_idid idid;
};
```

`void bai_init_radau5_result (struct bai_radau5_result *result)` [Function]

Initialize *result* tp the empty result. This function belongs to `bai_radau5_result.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_xyh_radau5_result (struct bai_radau5_result *result, double *x, double *y, double h)` [Function]

Set the fields of *result* using x, y and h. The `idid` field is set to `bai_idid_work_in_progress`. This function belongs to `bai_radau5_result.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_input_result_radau5_integration_state (struct bai_radau5_integration_state *R, struct bai_radau5_input *input, struct bai_radau5_result *result)` [Function]

Assign starting values to R fields using *input* and *result*. This function belongs to `bai_radau5_integration_state.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.5.5 Integration state

Description of type `bai_radau5_integration_state`. This data structure stores the variables used internally by the numerical integrator. This type declaration belongs to `bai_radau5_integration_state.h`.

```
struct bai_radau5_integration_state
{
  struct bai_radau5_input *input;
  struct bai_radau5_result *result;
  ba0_int_p tol_alloc;
  double *rtol;
  double *atol;
  struct bai_radau5_stats stats;
  struct bai_radau5_step_control step_control;
  struct bai_radau5_problem_structure problem_structure;
  struct bai_radau5_work_arrays work;
  struct bai_radau5_core_state core;
  struct bai_radau5_dense_output_state dense;
  struct bai_radau5_linear_layout linear;
};
```

`void bai_init_radau5_integration_state (struct bai_radau5_integration_state *R)` [Function]

Initialize R to the empty integration state. This function belongs to `bai_radau5_integration_state.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_input_result_radau5_integration_state (struct bai_radau5_integration_state *R, struct bai_radau5_input *input, struct bai_radau5_result *result)` [Function]

Assign starting values to R fields using *input* and *result*. This function belongs to `bai_radau5_integration_state.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_step_radau5_integration_state (struct bai_radau5_integration_state *R)` [Function]

Update R in order to perform one integration step. The integration process should be continued as long as the `result->idid` field of R is equal to `bai_idid_work_in_progress`. This function belongs to `bai_radau5_integration_state.c`. It is run by 10 tests, including 10 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_radau5_push_xend_integration_state (struct bai_radau5_integration_state *R, double xend)` [Function]

Set the endpoint before the first step, or extend a successfully completed integration to *xend*. Before the first step, *xend* may replace a dummy endpoint in the same integration direction. After completion, it must lie strictly beyond the current endpoint in the established direction. The next call to `bai_step_radau5_integration_state` resumes the integration. This function belongs to `bai_radau5_integration_state.c`. It is run by 6 tests, including 6 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.5.6 Dense output

Description of type `bai_radau5_dense_output_state`. This data structure is used for computing collocation polynomials. This type declaration belongs to `bai_radau5_dense_output_state.h`.

```
struct bai_radau5_dense_output_state
{
// double xout;
// the following fields are common to each accepted step
  ba0_int_p nn;
  ba0_int_p nn2;
  ba0_int_p nn3;
  ba0_int_p nn4;
  double c1m1;
  double c2m1;
// the following fields vary at each accepted step
  double xosol;                 // old xsol - the starting side of the interval
  double xsol;                  // new xsol - the ending side of the interval
  double hsol;                  // xsol - xosol (may be negative)
  double *coeffs;               // 4*n values; actually points to work.cont
};
```

`void bai_init_radau5_dense_output_state (struct bai_radau5_dense_output_state *dense)` [Function]

Initialize *dense* to the empty state. This function belongs to `bai_radau5_dense_output_state.c`. It is run by 6 tests, including 6 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`struct bai_radau5_dense_output_state *` [Function]
`bai_new_radau5_dense_output_state (void)`

Allocate a new state, initialize it and return it. This function belongs to `bai_radau5_dense_output_state.c`. It is run by 6 tests, including 6 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_radau5_dense_output_state (struct` [Function]
`bai_radau5_dense_output_state *dst, struct bai_radau5_dense_output_state *src)`

Assign *src* to *dst*. The `coeffs` field is duplicated. This function belongs to `bai_radau5_dense_output_state.c`. It is run by 6 tests, including 6 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`double` [Function]
`bai_eval_collocation_polynomial_radau5_dense_output_state (struct bai_radau5_dense_output_state *dense, ba0_int_p i, double x)`

This function belongs to `bai_radau5_dense_output_state.c`. It is run by 5 tests, including 5 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_append_tableof_radau5_dense_output_state (struct` [Function]
`bai_tableof_radau5_dense_output_state *T, struct bai_radau5_dense_output_state *dense)`

This function belongs to `bai_radau5_dense_output_state.c`. It is run by 6 tests, including 6 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`ba0_int_p bai_seek_tableof_radau5_dense_output_state` [Function]
`(struct bai_tableof_radau5_dense_output_state *T, double x)`

Return the index in T of a dense output state whose interval contains x. Return `BA0_NOT_AN_INDEX` if not found. This function belongs to `bai_radau5_dense_output_state.c`. It is run by 1 tests, including 1 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.5.7 Restore point

Description of type `bai_radau5_restore_point`. This data structure permits to store values associated to some integration point in order to restart the process at this point with a different step size. This type declaration belongs to `bai_radau5_restore_point.h`.

```
struct bai_radau5_restore_point
{
  ba0_int_p n;
/*
 * Current accepted solution
 */
  double x;
  double *y;                    /* n values */
/*
 * Previous-step data used to initialize Newton iterations
 */
  double hold;
  double xold;
  double *cont;                 /* 4*n values */
```

```
/*
 * Step-size controller history
 */
  double hacc;
  double erracc;
  double faccon;
  bool first;
/*
 * Dense-output metadata
 */
  double xsol;
  double hsol;
  double xosol;
  ba0_int_p nrsol;
  ba0_int_p nsolu;
/*
 * Failure history
 */
  ba0_int_p nsing;
/*
 * Diagnostics, if they must also roll back
 */
  double e1sing;
  double e2sing;
  double e1sold;
  double e2sold;
/*
 * Accounting, if restarted work must disappear from statistics
 */
  struct bai_radau5_stats stats;
};
```

`void bai_init_radau5_restore_point (struct bai_radau5_restore_point *point)` [Function]

Initialize *point* to the empty restore point. This function belongs to `bai_radau5_restore_point.c`. It is run by 6 tests, including 6 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_realloc_radau5_restore_point (struct bai_radau5_restore_point *point, ba0_int_p n)` [Function]

Reallocate the fields of *point* so that they can be used for systems of dimension n. This function belongs to `bai_radau5_restore_point.c`. It is run by 6 tests, including 6 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_state_radau5_restore_point (struct bai_radau5_restore_point *point, struct bai_radau5_integration_state *R)` [Function]

Assign to *point* copies of the fields of R which are necessary to restart the integration process from the current point. This function belongs to `bai_radau5_restore_point.c`. It is run by 6 tests, including 6 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_reset_for_restart_radau5_restore_point (struct bai_radau5_integration_state *R, double h, struct bai_radau5_restore_point *point)` [Function]

Reset the fields of R previously stored in *point* in order to restart the integration process with a new stepsize h. This function belongs to `bai_radau5_restore_point.c`. It is not run by any test.

`void bai_reset_as_completed_radau5_restore_point (struct bai_radau5_integration_state *R, struct bai_radau5_restore_point *point)` [Function]

Restore R from *point* and mark it as successfully completed at that point. This discards the state accumulated by an unsuccessful trial step, while retaining the last accepted solution as the endpoint of the integration. This function belongs to `bai_radau5_restore_point.c`. It is not run by any test.

10.5.8 Session

Description of type `bai_radau5_session`. This data structure bundles a few structures used by the RADAU5 numerical integrator which was primarily designed and implemented as a purely numerical software by E. Hairer and G. Wanner. See [HW96, section IV.8] and `https://www.unige.ch/~hairer/software.html`.

The `B` field contains the variable bindings associated to the session. The `input` field contains input data (including a pointer to `B`). The `result` field contains output data. Its subfield `x` and `y` point to storage areas located in the `B`. The `state` field contains state variables used by the integrator. This type declaration belongs to `bai_radau5_session.h`.

```
struct bai_radau5_session
{
  struct bai_variable_bindings B;
  struct bai_radau5_input input;
  struct bai_radau5_result result;
  struct bai_radau5_integration_state state;
};
```

`void bai_init_radau5_session (struct bai_radau5_session *S)` [Function]

Initialize S to the empty session. This function belongs to `bai_radau5_session.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_dtrpyx_radau5_session (struct bai_radau5_session *S, double x0, double xend, double h0, struct bai_radau5_tolerances *tol, struct bad_regchain *C, struct ba0_point_double *csts, struct bav_tableof_symbol *Y, struct baz_prolongation_pattern *Ybar, struct bav_symbol *x)` [Function]

This function belongs to `bai_radau5_session.c`. It is run by 11 tests, including 11 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_fprintf_tableof_variable_radau5_session (FILE *f, struct bav_tableof_variable *vars, struct bai_radau5_session *S, double x)` [Function]

Print one line in *f*. The printed line contains one floating point value per variable in *vars*. These variables must be recorded in the `struct bai_variable_bindings`

subfield of S. The evaluation is performed using the collocation polynomial stored in `S->state.dense`. Exception `BAO_ERRALG` is raised if x does not belong to the range defined by `xosol` and `xsol`. This function belongs to `bai_radau5_session.c`. It is run by 4 tests, including 4 in the `bai` library and 0 in the `DifferentialAlgebra` package.

10.5.9 Solution

Description of type `bai_radau5_idid`. This data type corresponds to the possible return values of the `bai_step_radau5_integration_state` function. This type declaration belongs to `bai_radau5_result.h`.

```
enum bai_radau5_idid
{
  bai_idid_work_in_progress = 0,
  bai_idid_computation_successful = 1,
  bai_idid_input_is_inconsistent = -1,
  bai_idid_larger_nmax_is_needed = -2,
  bai_idid_step_size_becomes_too_small = -3,
  bai_idid_matrix_is_repeatedly_singular = -4
};
```

Description of type `bai_radau5_solution`. This data structure aims at permitting the evaluation of derivatives at any value of the independent variable. The values of the derivatives are supposed to be obtained from some system plus some x_0, by means of numerical integration. For this, it contains two `struct bai_radau5_session` as subfields. The `forward` field is dedicated to the integration for $x > x_0$. The `backward` field is dedicated to the integration for $x < x_0$. The tables `forward_history` and `backward_history` store the content of `bai_radau5_dense_output_state` structures, after each accepted step. They aim at allowing evaluation by means of the collocation polynomial. This type declaration belongs to `bai_radau5_solution.h`.

```
struct bai_radau5_solution
{
  struct bai_radau5_session forward;
  struct bai_tableof_radau5_dense_output_state forward_history;
  struct bai_radau5_session backward;
  struct bai_tableof_radau5_dense_output_state backward_history;
};
```

`void bai_init_radau5_solution (struct bai_radau5_solution *S)` [Function]

Initialize S to the empty solution. This function belongs to `bai_radau5_solution.c`. It is run by 7 tests, including 7 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`void bai_set_dtrpyx_radau5_solution (struct bai_radau5_solution *S, double x0, struct bai_radau5_tolerances *tol, struct bad_regchain *C, struct ba0_point_double *csts, struct bav_tableof_symbol *Y, struct baz_prolongation_pattern *Ybar, struct bav_symbol *x)` [Function]

Complete the initialization of S using the other arguments. This function belongs to `bai_radau5_solution.c`. It is run by 7 tests, including 7 in the `bai` library and 0 in the `DifferentialAlgebra` package.

`double bai_eval_radau5_solution (struct bai_radau5_solution *S, struct bav_variable *v, double x, enum bai_radau5_idid *idid)` [Function]

When successful, return v evaluated at x. The derivative v must occur in S as a constant variable, the independent variable or a non extra computed variable. If non-zero, the argument *idid* receives `bai_idid_computation_successful`. The evaluation, which may require some numerical integration process, may also possibly fail. In such a case, *idid* receives a negative value and the returned value is `nan`. This function belongs to `bai_radau5_solution.c`. It is run by 7 tests, including 7 in the `bai` library and 0 in the `DifferentialAlgebra` package.

Acknowledgements

François Boulier wrote the initial BLAD libraries. Several persons have contributed to them in different ways. Here is a list.

- Jean-Charles Faugère and Fabrice Rouillier contributed indirectly by providing the memory management schemes carried out in their own software.
- François Lemaire and Marc Moreno Maza contributed indirectly to the design of many functions. Discussions and comparative tests w.r.t. the MAPLE RegularChains package were most helpful.
- Edgardo Cheb-Terrab, Jürgen Gerhard and Clare So supervised the development of the DifferentialAlgebra MAPLE package. They more or less directly suggested many developments and enhancements. The BLAD libraries benefited a lot from their works.
- Alexey Ovchinnikov helped porting the libraries on Mac OS X (Darwin) and helped fixing bugs.
- Anthony Hebig, Sylvain Neut and Markus Lange-Hegermann for bug fixing.
- Gabriel Dos Reis for his comments.
- Markus Rosenkranz organized a LTCC course on the DifferentialAlgebra MAPLE package. He exhibited a bug in the implementation of the Low Power Theorem.

Function Index

ba0_abs_interval_mpq 76
ba0_add_dictionary 87
ba0_add_dictionary_int_p 88
ba0_add_dictionary_string 89
ba0_add_dictionary_typed_string 91
ba0_add_interval_mpq 76
ba0_add_interval_mpq_mpq 76
ba0_alloc 23
ba0_alloc_mark 25
ba0_allocated_size 22
ba0_append_table_list 55
ba0_append_tableof_double_listof_double ... 58
ba0_append_tableof_int_p_listof_int_p 57
ba0_are_disjoint_interval_mpq 75
ba0_are_equal_interval_mpq 75
ba0_assoc_point 64
ba0_assoc_point_double 67
ba0_assoc_point_int_p 66
ba0_atof 72
ba0_Bareiss_matrix 60
ba0_bsearch_point 65
ba0_bsearch_point_double 67
ba0_bsearch_point_int_p 66
ba0_cancel_PFE_settings 18
ba0_ceil_align 22
ba0_cell_index_mark 25
ba0_clear_analex 47
ba0_clear_cells_stack 23
ba0_clear_one_cell_stack 23
ba0_clear_stack 23
ba0_clear_stack_of_stacks 25
ba0_compatible_indices_range_indexed_group 85
ba0_cons_list 52
ba0_cons_listof_double 53
ba0_cons_listof_int_p 52
ba0_contains_zero_interval_mpq 75
ba0_current_stack 23
ba0_define_format 32
ba0_define_format_with_sizelt 32
ba0_delete_point 64
ba0_delete_point_double 67
ba0_delete_point_int_p 65
ba0_delete_table 54
ba0_delete_tableof_double 58
ba0_delete_tableof_int_p 56
ba0_div_interval_mpq 77
ba0_div_interval_mpq_mpq 78
ba0_div_mpq_interval_mpq 78
ba0_domain_interval_mpq 73
ba0_domain_mint_hp 70
ba0_domain_mpzm 70
ba0_element_interval_mpq 76
ba0_exception_is_raised 41
ba0_exception_is_set 41
ba0_fit_range_indexed_group 85
ba0_fprintf 50
ba0_free 20
ba0_fscanf 50
ba0_fscanf2 50
ba0_garbage 27
ba0_get_char 45
ba0_get_context_analex 45
ba0_get_counter_analex 48
ba0_get_dictionary 87
ba0_get_dictionary_int_p 88
ba0_get_dictionary_string 89
ba0_get_dictionary_typed_string 91
ba0_get_format 32
ba0_get_settings_analex 46
ba0_get_settings_common 19
ba0_get_settings_gmp 21
ba0_get_settings_interrupt 19
ba0_get_settings_memory_functions 20
ba0_get_settings_no_oom 18
ba0_get_settings_no_oot 17
ba0_get_settings_range_indexed_group 83
ba0_get_settings_stack 22
ba0_get_settings_value 62
ba0_get_token_analex 48
ba0_get_version 11
ba0_has_double_format_component 33
ba0_has_empty_trailing_indices_indexed_string 81
ba0_has_int_p_format_component 33
ba0_has_numeric_trailing_indices_indexed_string 81
ba0_has_trailing_indices_indexed_string ... 81
ba0_in_stack 24
ba0_indexed_string_to_string 81
ba0_init_analex 47
ba0_init_dictionary 86
ba0_init_dictionary_int_p 87
ba0_init_dictionary_string 89
ba0_init_dictionary_typed_string 90
ba0_init_indexed_string 80
ba0_init_interval_mpq 73
ba0_init_matrix 59
ba0_init_matrixof_double 61
ba0_init_matrixof_int_p 60
ba0_init_one_cell_stack 23
ba0_init_point 64
ba0_init_point_double 66
ba0_init_point_int_p 65
ba0_init_range_indexed_group 84
ba0_init_stack 22
ba0_init_stack_of_stacks 25
ba0_init_table 53
ba0_init_tableof_double 57
ba0_init_tableof_int_p 56

ba0_init_value 62
ba0_init_value_double 63
ba0_init_value_int_p 62
ba0_initialize_format 32
ba0_insert_table 54
ba0_insert_tableof_double 58
ba0_insert_tableof_int_p 57
ba0_insert2_list 52
ba0_intersect_interval_mpq 75
ba0_is_closed_interval_mpq 74
ba0_is_double_format_component 33
ba0_is_empty_interval_mpq 74
ba0_is_garbaged_format_component 33
ba0_is_int_p_format_component 33
ba0_is_less_interval_mpq 75
ba0_is_negative_interval_mpq 75
ba0_is_nonnegative_interval_mpq 75
ba0_is_nonpositive_interval_mpq 75
ba0_is_one_interval_mpq 74
ba0_is_open_interval_mpq 74
ba0_is_plain_string_range_indexed_group 85
ba0_is_positive_interval_mpq 75
ba0_is_sorted_point 64
ba0_is_sorted_point_double 67
ba0_is_sorted_point_int_p 66
ba0_is_subset_interval_mpq 76
ba0_is_unbounded_interval_mpq 74
ba0_is_zero_interval_mpq 74
ba0_isatty_input 45
ba0_isinf 72
ba0_isnan 72
ba0_length_list 52
ba0_length_listof_double 53
ba0_length_listof_int_p 53
ba0_log2_int_p 68
ba0_malloc 20
ba0_max_alloc_stack 24
ba0_member_interval_mpq 75
ba0_member_table 55
ba0_member_tableof_double 58
ba0_member_tableof_int_p 57
ba0_member2_table 55
ba0_member2_tableof_double 58
ba0_member2_tableof_int_p 57
ba0_member2_tableof_string 79
ba0_memory_left_in_cell 24
ba0_middle_interval_mpq 76
ba0_middle_interval_mpq_double 76
ba0_mint_hp_module_set 70
ba0_move_from_tail_table 55
ba0_move_from_tail_tableof_double 59
ba0_move_from_tail_tableof_int_p 57
ba0_move_to_tail_table 55
ba0_move_to_tail_tableof_double 58
ba0_move_to_tail_tableof_int_p 57
ba0_mpzm_module_mul 71
ba0_mpzm_module_pow_ui 71
ba0_mpzm_module_set 71
ba0_mpzm_module_set_ui 71
ba0_mul_interval_mpq 77
ba0_mul_interval_mpq_mpq 77
ba0_mul_interval_mpq_si 77
ba0_mul_interval_mpq_ui 77
ba0_neg_interval_mpq 76
ba0_nested_format_component 33
ba0_new_double 72
ba0_new_indexed_string 80
ba0_new_interval_mpq 73
ba0_new_matrix 59
ba0_new_matrixof_double 61
ba0_new_matrixof_int_p 61
ba0_new_mpq 71
ba0_new_mpz 69
ba0_new_mpzm 70
ba0_new_printf 50
ba0_new_range_indexed_group 85
ba0_new_string 78
ba0_new_table 54
ba0_new_tableof_double 58
ba0_new_tableof_int_p 56
ba0_new_value 62
ba0_new_value_double 63
ba0_new_value_int_p 63
ba0_not_a_string 78
ba0_output_counter 43
ba0_persistent_malloc 20
ba0_pow_interval_mpq 77
ba0_pow_interval_mpq_mpq 77
ba0_printf 50
ba0_printf_bool 69
ba0_printf_double 72
ba0_printf_hexint_p 68
ba0_printf_indexed_string 81
ba0_printf_int_p 68
ba0_printf_interval_mpq 78
ba0_printf_mpq 72
ba0_printf_mpz 69
ba0_printf_mpzm 71
ba0_printf_range_indexed_group 86
ba0_printf_string 79
ba0_pull_exception 41
ba0_pull_exception_extra_stack 40
ba0_pull_stack 25
ba0_pull_subs_dict_analex 46
ba0_push_another_stack 25
ba0_push_exception 41
ba0_push_exception_extra_stack 40
ba0_push_stack 25
ba0_push_subs_dict_analex 46
ba0_put_char 43
ba0_put_hexint_p 44
ba0_put_int_p 44
ba0_put_string 43
ba0_raise_exception 42
ba0_raise_exception2 42
ba0_range_mark 26

ba0_re_malloc_table 54
ba0_re_malloc_tableof_double 58
ba0_re_malloc_tableof_int_p 56
ba0_realloc_matrix 60
ba0_realloc_matrixof_double 61
ba0_realloc_matrixof_int_p 61
ba0_realloc_point 64
ba0_realloc_point_double 67
ba0_realloc_point_int_p 65
ba0_realloc_table 54
ba0_realloc_tableof_double 58
ba0_realloc_tableof_int_p 56
ba0_realloc2_matrix 60
ba0_realloc2_table 54
ba0_record 26
ba0_record_analex 47
ba0_record_gmp_memory_functions 69
ba0_record_input 44
ba0_record_output 43
ba0_reset_all_settings 17
ba0_reset_analex 47
ba0_reset_cell_stack 23
ba0_reset_dictionary 86
ba0_reset_dictionary_int_p 88
ba0_reset_dictionary_string 89
ba0_reset_dictionary_typed_string 90
ba0_reset_exception 41
ba0_reset_exception_extra_stack 40
ba0_reset_indexed_string 80
ba0_reset_input 44
ba0_reset_output 43
ba0_reset_point 64
ba0_reset_range_indexed_group 84
ba0_reset_stack 23
ba0_reset_stack_of_stacks 25
ba0_reset_subs_dict_analex 46
ba0_reset_table 53
ba0_reset_tableof_double 57
ba0_reset_tableof_int_p 56
ba0_restart 18
ba0_restore 26
ba0_restore_analex 48
ba0_restore_gmp_memory_functions 69
ba0_restore_input 44
ba0_restore_output 43
ba0_restore_PFE_settings 18
ba0_reverse_list 52
ba0_rotate_cells 24
ba0_scanf 49
ba0_scanf_bool 68
ba0_scanf_double 72
ba0_scanf_hexint_p 68
ba0_scanf_indexed_string 82
ba0_scanf_indexed_string_as_a_string 81
ba0_scanf_indexed_string_with_counter 82
ba0_scanf_int_p 68
ba0_scanf_interval_mpq 78
ba0_scanf_mpq 71
ba0_scanf_mpz 69
ba0_scanf_mpzm 71
ba0_scanf_printf 50
ba0_scanf_range_indexed_group 85
ba0_scanf_string 79
ba0_scanf2 49
ba0_set_analex_FILE 48
ba0_set_analex_string 48
ba0_set_dictionary 87
ba0_set_dictionary_int_p 88
ba0_set_dictionary_string 89
ba0_set_dictionary_typed_string 91
ba0_set_indexed_string 81
ba0_set_input_FILE 44
ba0_set_input_string 44
ba0_set_interval_mpq 74
ba0_set_interval_mpq_double 73
ba0_set_interval_mpq_mpq 74
ba0_set_interval_mpq_si 73
ba0_set_interval_mpq_type_mpq 74
ba0_set_interval_mpq_ui 73
ba0_set_matrix 59
ba0_set_matrixof_double 61
ba0_set_matrixof_int_p 61
ba0_set_output_counter 43
ba0_set_output_FILE 43
ba0_set_output_string 43
ba0_set_point 64
ba0_set_point_double 67
ba0_set_point_int_p 65
ba0_set_point_list 64
ba0_set_range_indexed_group 83, 85
ba0_set_range_indexed_group_string 85
ba0_set_range_indexed_group_with_tableof_string 83
ba0_set_settings_analex 45
ba0_set_settings_common 19
ba0_set_settings_gmp 21
ba0_set_settings_interrupt 19
ba0_set_settings_memory_functions 20
ba0_set_settings_no_oom 17
ba0_set_settings_no_oot 17
ba0_set_settings_range_indexed_group 83
ba0_set_settings_stack 22
ba0_set_settings_value 62
ba0_set_table 54
ba0_set_table_list 55
ba0_set_tableof_double 58
ba0_set_tableof_int_p 57
ba0_set_tableof_mpz 69
ba0_set_tableof_range_indexed_group_with_tableof_string 84
ba0_set_tableof_string 79
ba0_set_tableof_string_tableof_range_indexed_group 84
ba0_set_tableof_tableof_int_p 68
ba0_set_tableof_tableof_mpz 69
ba0_set2_table 54

ba0_sign_token_analex 49
ba0_sizeof_dictionary 87
ba0_sizeof_dictionary_int_p 88
ba0_sizeof_dictionary_string 89
ba0_sizeof_dictionary_typed_string 90
ba0_sizeof_range_indexed_group 83
ba0_sizeof_table 54
ba0_sizeof_tableof_double 58
ba0_sizeof_tableof_int_p 56
ba0_sizeof_tableof_range_indexed_group 84
ba0_sizeof_tableof_string 79
ba0_sizeof_tableof_tableof_int_p 68
ba0_sort_list 52
ba0_sort_point 64
ba0_sort_point_double 67
ba0_sort_point_int_p 66
ba0_sort_table 55
ba0_sort2_list 52
ba0_spaces_before_token_analex 49
ba0_sprintf 50
ba0_sscanf 49
ba0_sscanf2 50
ba0_strcasecmp 79
ba0_strcat 78
ba0_strdup 78
ba0_stripped_indexed_string_to_string 81
ba0_strncasecmp 79
ba0_sub_interval_mpq 77
ba0_sub_interval_mpq_mpq 77
ba0_sub_mpq_interval_mpq 77
ba0_t1_alloc 24
ba0_t2_alloc 24
ba0_terminate 19
ba0_type_token_analex 49
ba0_unget_char 45
ba0_unget_given_token_analex 48
ba0_unget_token_analex 48
ba0_unique_table 56
ba0_value_token_analex 49
ba0_wang_mint_hp 70
ba0_wang_mpzm 71
ba0_which_stack 24
ba0_width_interval_mpq 76
ba0_width_interval_mpq_double 76
ba0_write_context_analex 45
BA0_CANCEL_EXCEPTION 41
BA0_CATCH 41
BA0_CERR 40
BA0_ENDTRY 41
BA0_MAT 59
BA0_RAISE_EXCEPTION 41
BA0_RAISE_EXCEPTION2 42
BA0_RAISE_PARSER_EXCEPTION 41
BA0_TRY 41
bad_add_edge_novar_splitting_tree 237
bad_add_edge_splitting_tree 237
bad_address_property_quench_map 218
bad_append_intersectof_regchain 214
bad_append_intersectof_regchain_regchain . 214
bad_apply_dimension_lower_bound_splitting_control 244
bad_base_field_generators 221
bad_check_regularity_polynom_mod_regchain 228
bad_clear_property_attchain 205
bad_clear_property_intersectof_regchain . 214
bad_clear_property_regchain 208
bad_codimension_regchain 209
bad_complete_quadruple 248
bad_defines_a_differential_ideal_attchain 206
bad_defines_a_differential_ideal_regchain 209
bad_defines_a_prime_ideal_attchain 206
bad_defines_a_prime_ideal_regchain 209
bad_delete_from_listof_polynom_mpz 246
bad_delta_polynom_critical_pair 233
bad_depends_on_leader_of_regchain 210
bad_dot_splitting_tree 237
bad_double_penalty_selection_strategy 235
bad_ensure_nonzero_initial_mod_regchain . 225
bad_equal_attchain 206
bad_Euclid_mod_regchain 227
bad_extend_quadruple_regchain 245
bad_extend_regchain 208
bad_fast_primality_test_intersectof_regchain 215
bad_fast_primality_test_regchain 211
bad_first_index_quench_map 218
bad_first_quadruple 250
bad_fully_reduced_polynom_quench_map 218
bad_get_number_quadruple 245
bad_get_number_regchain 207
bad_get_regchain_intersectof_regchain 215
bad_get_settings_preparation 251
bad_get_settings_reduction 224
bad_get_settings_regularize 227
bad_handle_splitting_exceptions_regchain . 220
bad_has_multiplicity_typeof_splitting_edge 241
bad_has_property_attchain 206
bad_has_property_intersectof_regchain 214
bad_has_property_regchain 209
bad_has_var_typeof_splitting_edge 241
bad_inactivate_property_quench_map 217
bad_inequations_regchain 212
bad_init_attchain 205
bad_init_base_field 221
bad_init_critical_pair 232
bad_init_from_complete_quench_map 217
bad_init_intersectof_regchain 213
bad_init_preparation_equation 251
bad_init_quadruple 244
bad_init_quench_map 217
bad_init_regchain 207
bad_init_selection_strategy 234

bad_init_set_quench_map 217
bad_init_splitting_control 243
bad_init_splitting_edge 241
bad_init_splitting_tree 236
bad_init_splitting_vertex 239
bad_init_stats 204
bad_insert_in_listof_polynom_mpz 246
bad_insert_in_P_quadruple 245
bad_insert_in_S_quadruple 245
bad_insert_polynom_in_regchain 208
bad_intersect_attchain 205
bad_invert_polynom_mod_regchain 229
bad_invert_product_mod_regchain 229
bad_is_a_compatible_base_field 222
bad_is_a_compatible_regchain 211
bad_is_a_listof_rejected_critical_pair ... 233
bad_is_a_partially_reduced_polynom_wrt_
regchain 225
bad_is_a_property_attchain 206
bad_is_a_reduced_to_zero_polynom_by_
regchain 226
bad_is_a_reducible_polynom_by_regchain ... 225
bad_is_a_reducible_product_by_regchain ... 225
bad_is_a_reduction_critical_pair 233
bad_is_a_simpler_critical_pair 233
bad_is_an_algebraic_critical_pair 233
bad_is_an_already_satisfied_property_
quench_map 218
bad_is_derivative_of_leader_of_regchain . 210
bad_is_differential_base_field 221
bad_is_explicit_regchain 211
bad_is_first_vertex_splitting_tree 236
bad_is_included_regchain 212
bad_is_leader_of_regchain 210
bad_is_orthonomic_regchain 211
bad_is_rank_of_regchain 210
bad_is_solved_regchain 210
bad_is_zero_regchain 211
bad_ith_vertex_splitting_tree 236
bad_leader_symbol_splitting_edge 241
bad_leaders_of_regchain 210
bad_low_power_theorem_condition_to_be_a_
component 252
bad_low_power_theorem_simplify_intersectof_
regchain 252
bad_mark_indets_quadruple 245
bad_mark_indets_regchain 207
bad_member_nonzero_polynom_base_field 222
bad_member_polynom_base_field 222
bad_member_product_base_field 223
bad_member_variable_base_field 222
bad_merge_thetas_leaders_splitting_
vertex 239
bad_merge_thetas_leaders_vertex_splitting_
tree 237
bad_new_base_field 221
bad_new_critical_pair 232
bad_new_critical_pair_polynom_mpz 232
bad_new_intersectof_regchain 213
bad_new_preparation_equation 251
bad_new_quadruple 245
bad_new_regchain 207
bad_new_selection_strategy 234
bad_new_splitting_control 243
bad_new_splitting_edge 241
bad_new_splitting_tree 236
bad_new_splitting_vertex 239
bad_next_number_splitting_tree 236
bad_normal_form_handling_exceptions_
ratfrac_mod_regchain 230
bad_normal_form_polynom_mod_regchain 230
bad_normal_form_ratfrac_mod_intersectof_
regchain 230
bad_normal_form_ratfrac_mod_regchain 230
bad_normalized_polynom_quench_map 218
bad_number_of_elements_over_base_field_
regchain 222
bad_ordering_eliminating_leaders_of_
regchain 211
bad_pardi 250
bad_partially_reduced_polynom_quench_map . 218
bad_pick_and_remove_quadruple 247
bad_preparation_congruence 252
bad_preprocess_equation_quadruple 246
bad_primitive_polynom_quench_map 218
bad_printf_base_field 223
bad_printf_critical_pair 234
bad_printf_intersectof_regchain 216
bad_printf_intersectof_regchain_
equations 216
bad_printf_preparation_equation 252
bad_printf_quadruple 249
bad_printf_quench_map 219
bad_printf_regchain 212
bad_printf_regchain_equations 212
bad_printf_splitting_edge 242
bad_printf_splitting_tree 237
bad_printf_splitting_vertex 240
bad_printf_stats 204
bad_product_of_leading_degrees_regchain . 211
bad_properties_attchain 206
bad_properties_intersectof_regchain 215
bad_properties_regchain 209
bad_pseudo_divided_polynom_quench_map 219
bad_quench_and_handle_exceptions_
regchain 219
bad_quench_regchain 219
bad_realloc_intersectof_regchain 213
bad_realloc_regchain 207
bad_reduce_easy_polynom_by_regchain 225
bad_reduce_polynom_by_regchain 225
bad_reduce_product_by_regchain 226
bad_reduced_form_polynom_mod_intersectof_
regchain 231
bad_reduced_form_polynom_mod_regchain 231
bad_reg_characteristic_quadruple 248

bad_reg_characteristic_regchain 229
bad_remove_product_factors_base_field 223
bad_remove_redundant_components_intersectof_regchain 215
bad_report_simplification_of_inequations_quadruple 246
bad_reset_all_settings 202
bad_reset_attchain 205
bad_reset_intersectof_regchain 213
bad_reset_regchain 207
bad_reset_splitting_tree 236
bad_restart 202
bad_resultant_mod_regchain 228
bad_Rosenfeld_Groebner 249
bad_scanf_base_field 223
bad_scanf_critical_pair 234
bad_scanf_intersectof_pretend_regchain ... 216
bad_scanf_intersectof_regchain 215
bad_scanf_pretend_regchain 212
bad_scanf_quadruple 248
bad_scanf_regchain 212
bad_scanf_splitting_edge 242
bad_scanf_splitting_tree 237
bad_scanf_splitting_vertex 239
bad_set_all_properties_quench_map 217
bad_set_attchain 205
bad_set_automatic_properties_attchain 205
bad_set_automatic_properties_intersectof_regchain 214
bad_set_automatic_properties_regchain 209
bad_set_base_field 221
bad_set_base_field_generators_and_relations 222
bad_set_base_field_relations_properties . 221
bad_set_critical_pair 232
bad_set_critical_pair_polynom_mpz 233
bad_set_DenefLipshitz_splitting_control . 243
bad_set_dimension_lower_bound_splitting_control 243
bad_set_first_leaf_only_splitting_control 243
bad_set_first_vertex_splitting_tree 236
bad_set_intersectof_regchain 213
bad_set_intersectof_regchain_regchain 214
bad_set_next_number_quadruple 245
bad_set_number_quadruple 245
bad_set_number_regchain 207
bad_set_penalty_selection_strategy 235
bad_set_preparation_equation_polynom 252
bad_set_properties_attchain 205
bad_set_properties_intersectof_regchain . 214
bad_set_properties_regchain 209
bad_set_property_attchain 205
bad_set_property_intersectof_regchain 214
bad_set_property_quench_map 217
bad_set_property_regchain 208
bad_set_quadruple 245
bad_set_regchain 207
bad_set_regchain_tableof_polynom_mpz 208
bad_set_regchain_tableof_ratfrac_mpz 208
bad_set_settings_preparation 251
bad_set_settings_reduction 224
bad_set_settings_regularize 227
bad_set_splitting_control 243
bad_set_splitting_edge 242
bad_set_splitting_tree 236
bad_set_splitting_vertex 239
bad_set_strategy_selection_strategy 234
bad_set_tsdvm_splitting_edge 241
bad_shapeof_splitting_vertex_to_string ... 239
bad_simplify_and_store_in_P_quadruple 246
bad_simplify_and_store_in_S_quadruple 247
bad_sizeof_regchain 212
bad_sort_intersectof_regchain 215
bad_sort_regchain 210
bad_split_on_factors_of_equations_quadruple 247
bad_switch_ring_regchain 213
bad_terminate 202
bad_thetas_and_leaders_critical_pair 233
bad_typeof_splitting_edge_to_string 240
bai_add_computed_variable_bindings 273
bai_add_constant_variable_bindings 273
bai_add_extra_computed_variable_bindings . 273
bai_add_point_constant_variable_bindings . 273
bai_add_regchain_computed_variable_bindings 273
bai_add_regchain_constant_variable_bindings 274
bai_add_regchain_variable_bindings 274
bai_add_spx_variable_bindings 274
bai_append_tableof_radau5_dense_output_state 282
bai_eval_collocation_polynomial_radau5_dense_output_state 282
bai_eval_radau5_solution 286
bai_eval_recursive_polynomial 276
bai_fprintf_tableof_variable_radau5_session 284
bai_init_radau5_dense_output_state 281
bai_init_radau5_input 279
bai_init_radau5_integration_state 280
bai_init_radau5_recursive_polynomial_data 277
bai_init_radau5_restore_point 283
bai_init_radau5_result 280
bai_init_radau5_session 284
bai_init_radau5_solution 285
bai_init_radau5_tolerances 278
bai_init_recursive_polynomial 276
bai_init_variable_bindings 274
bai_jacobian_f_polynom_mpz 275
bai_mass_f_regchain_polynom_mpz 275
bai_new_radau5_dense_output_state 282
bai_new_recursive_polynomial 276
bai_new_variable_bindings 274

bai_radau5_push_xend_integration_state... 281
bai_realloc_radau5_restore_point 283
bai_reset_as_completed_radau5_restore_point 284
bai_reset_for_restart_radau5_restore_point 284
bai_reset_initial_values_variable_bindings 274
bai_restart 271
bai_seek_tableof_radau5_dense_output_state 282
bai_set_double_coefficient 275
bai_set_double_radau5_tolerances 278
bai_set_dtrpyx_radau5_session 284
bai_set_dtrpyx_radau5_solution 285
bai_set_input_result_radau5_integration_state 279, 280, 281
bai_set_matrixof_polynom_mpz_matrixof_recursive_polynomial 276
bai_set_nxyh_radau5_initial_values 278
bai_set_polynom_coefficient 275
bai_set_polynom_mpz_recursive_polynomial. 277
bai_set_radau5_dense_output_state 282
bai_set_radau5_tolerances 279
bai_set_regchain_radau5_input 279
bai_set_regchain_radau5_recursive_polynomial_data 277
bai_set_state_radau5_restore_point 283
bai_set_tableof_polynom_mpz_tableof_recursive_polynomial 277
bai_set_variable_bindings 275
bai_set_variable_recursive_polynomial 277
bai_set_xyh_radau5_result 280
bai_step_radau5_integration_state 281
bai_terminate 271
bap_add_geobucket_COEFF 166
bap_add_polynom_COEFF 158
bap_add_polynom_numeric_COEFF 158
bap_addmulrk_polynom_COEFF 159
bap_append_creator_clot_COEFF 140
bap_append_creator_COEFF 175
bap_append_termstripper 146
bap_are_disjoint_polynom_COEFF 147
bap_begin_creator_clot_COEFF 140
bap_begin_creator_COEFF 175
bap_begin_creator_indexed_access 144
bap_begin_iterator_indexed_access 143
bap_begin_itercoeff_COEFF 171
bap_begin_itermon_clot_COEFF 139
bap_begin_itermon_COEFF 172
bap_Berlekamp_mint_hp 170
bap_Bezout_polynom_mpzm 169
bap_change_ordering_clot_COEFF 138
bap_change_ordering_termstripper 145
bap_change_variable_termstripper 145
bap_close_creator_clot_COEFF 141
bap_close_creator_COEFF 176
bap_close_creator_indexed_access 144
bap_close_itercoeff_COEFF 172
bap_close_itermon_COEFF 174
bap_coeff_and_reductum_polynom_COEFF 155
bap_coeff_itercoeff_COEFF 172
bap_coeff_itermon_clot_COEFF 140
bap_coeff_itermon_COEFF 173
bap_coeff_polynom_COEFF 154
bap_coeff2_polynom_COEFF 155
bap_coeftayl_polynom_mpzm 169
bap_comblin_polynom_COEFF 159
bap_compare_polynom_COEFF 151
bap_compare_rank_polynom_COEFF 152
bap_degree_polynom_COEFF 152
bap_denom_polynom_mpq 168
bap_depend_only_polynom_COEFF 151
bap_depend_polynom_COEFF 151
bap_diff_polynom_COEFF 164
bap_diff2_polynom_COEFF 164
bap_end_iterator_indexed_access 143
bap_end_itercoeff_COEFF 171
bap_end_itermon_clot_COEFF 139
bap_end_itermon_COEFF 172
bap_equal_polynom_COEFF 152
bap_equal_rank_polynom_COEFF 152
bap_equal_termanager 136
bap_Euclid_polynom_COEFF 162
bap_Euclidean_division_polynom_COEFF 162
bap_eval_to_numeric_at_point_int_p_polynom_COEFF 165
bap_eval_to_polynom_at_numeric_polynom_COEFF 165
bap_eval_to_polynom_at_point_int_p_polynom_COEFF 165
bap_eval_to_polynom_at_value_int_p_polynom_COEFF 165
bap_evalcoeff_at_point_int_p_polynom_COEFF 166
bap_expand_product_COEFF 179
bap_exponent_product_COEFF 179
bap_exquo_polynom_COEFF 161
bap_exquo_polynom_numeric_mpz 168
bap_exquo_polynom_product_COEFF 161
bap_exquo_polynom_term_COEFF 161
bap_exquo_product_polynom_COEFF 180
bap_extended_Euclid_polynom_COEFF 162
bap_goto_iterator_indexed_access 143
bap_goto_itermon_clot_COEFF 140
bap_goto_itermon_COEFF 173
bap_gt_rank_polynom_COEFF 152
bap_identity_termstripper 146
bap_index_iterator_indexed_access 143
bap_init_geobucket_COEFF 166
bap_init_indexed_access 142
bap_init_polynom_COEFF 148
bap_init_polynom_crk_COEFF 148
bap_init_polynom_one_COEFF 148
bap_init_power_COEFF 177
bap_init_product_COEFF 177

bap_init_product_zero_COEFF 177
bap_init_readonly_polynom_COEFF 148
bap_init_set_termanager 136
bap_init_set_termstripper 145
bap_init_termanager 135
bap_init_zipterm_array_termanager 136
bap_initial_and_reductum_polynom_COEFF ... 153
bap_initial_and_reductum2_polynom_COEFF . 154
bap_initial_polynom_COEFF 154
bap_initial2_polynom_COEFF 154
bap_involved_derivations_polynom_COEFF ... 156
bap_involved_parameters_polynom_COEFF 156, 165
bap_is_constant_polynom_COEFF 164
bap_is_derivative_minus_independent_polynom_COEFF 151
bap_is_factor_polynom_COEFF 160
bap_is_independent_polynom_COEFF 164
bap_is_numeric_factor_polynom_COEFF 160
bap_is_numeric_factor_polynom_mpz 168
bap_is_numeric_polynom_COEFF 150
bap_is_numeric_product_COEFF 178
bap_is_one_polynom_COEFF 150
bap_is_one_product_COEFF 178
bap_is_polynomial_with_constant_coefficients_COEFF 164
bap_is_rank_minus_monom_polynom_COEFF 151
bap_is_solved_constant_initial_polynom_COEFF 151
bap_is_solved_numeric_initial_polynom_COEFF 151
bap_is_univariate_polynom_COEFF 150
bap_is_variable_factor_polynom_COEFF 160
bap_is_variable_polynom_COEFF 150
bap_is_write_allable_creator_COEFF 175
bap_is_zero_clot_COEFF 138
bap_is_zero_polynom_COEFF 150
bap_is_zero_product_COEFF 178
bap_lcoeff_and_reductum_polynom_COEFF 155
bap_lcoeff_polynom_COEFF 155
bap_leader_polynom_COEFF 152
bap_leader_product_COEFF 179
bap_leading_degree_polynom_COEFF 153
bap_leading_term_polynom_COEFF 153
bap_lsr3_Ducos_polynom_COEFF 163
bap_lt_rank_polynom_COEFF 152
bap_mark_indets_listof_polynom_COEFF 157
bap_mark_indets_polynom_COEFF 157
bap_mark_indets_tableof_polynom_COEFF 157
bap_maxnorm_polynom_mpz 167
bap_minimal_total_rank_polynom_COEFF 153
bap_mods_polynom_mpzm 169
bap_mods_product_mpzm 169
bap_mul_geobucket_COEFF 167
bap_mul_geobucket_numeric_COEFF 166
bap_mul_polynom_COEFF 160
bap_mul_polynom_monom_COEFF 160
bap_mul_polynom_numeric_COEFF 159
bap_mul_polynom_term_COEFF 160
bap_mul_polynom_value_int_p_COEFF 159
bap_mul_polynom_variable_COEFF 160
bap_mul_product_COEFF 180
bap_mul_product_numeric_COEFF 180
bap_mul_product_polynom_COEFF 179
bap_mul_product_term_COEFF 180
bap_muldiv_Lazard_polynom_COEFF 163
bap_muldiv2_Lazard_polynom_COEFF 163
bap_muldiv3_Lazard_polynom_COEFF 163
bap_multi_Diophante_polynom_mpzm 170
bap_nbmon_polynom_COEFF 152
bap_neg_polynom_COEFF 159
bap_neg_product_COEFF 179
bap_new_clot_COEFF 138
bap_new_polynom_COEFF 148
bap_new_polynom_crk_COEFF 148
bap_new_polynom_one_COEFF 148
bap_new_power_COEFF 177
bap_new_product_COEFF 178
bap_new_product_zero_COEFF 178
bap_new_readonly_polynom_COEFF 148
bap_next_iterator_indexed_access 143
bap_next_itercoeff_COEFF 171
bap_next_itermon_clot_COEFF 139
bap_next_itermon_COEFF 173
bap_normal_numeric_primpart_polynom_mpz . 168
bap_normal_sign_polynom_mpz 167
bap_nsr2_Ducos_polynom_COEFF 162
bap_nsr3_Ducos_polynom_COEFF 163
bap_number_itermon_clot_COEFF 139
bap_numer_polynom_mpq 168
bap_numeric_content_polynom_mpz 167
bap_numeric_initial_one_polynom_COEFF 151
bap_numeric_initial_polynom_COEFF 153
bap_numeric_primpart_polynom_mpz 167
bap_outof_iterator_indexed_access 143
bap_outof_itercoeff_COEFF 171
bap_outof_itermon_clot_COEFF 139
bap_outof_itermon_COEFF 173
bap_physort_polynom_COEFF 147
bap_physort_product_COEFF 179
bap_polynom_mint_hp_to_mpzm 169
bap_polynom_mpq_to_mpz 167
bap_polynom_mpz_to_mint_hp 170
bap_polynom_mpz_to_mpq 168
bap_polynom_mpz_to_mpzm 169
bap_pow_polynom_COEFF 160
bap_pow_power_COEFF 177
bap_pow_product_COEFF 180
bap_pquo_polynom_COEFF 161
bap_prem_polynom_COEFF 161
bap_prev_iterator_indexed_access 143
bap_prev_itercoeff_COEFF 171
bap_prev_itermon_clot_COEFF 139
bap_prev_itermon_COEFF 173
bap_printf_polynom_COEFF 158
bap_printf_product_COEFF 180

bap_product_mpz_to_mpq 180
bap_pseudo_division_polynom_COEFF 161
bap_quorem_polynom_mpzm 169
bap_random_eval_polynom_mpz_to_mint_hp ... 170
bap_rank_polynom_COEFF 153
bap_read_iterator_indexed_access 144
bap_realloc_indexed_access 142
bap_realloc_product_COEFF 177
bap_reductum_itermon_COEFF 173
bap_reductum_polynom_COEFF 154
bap_reductum2_polynom_COEFF 154
bap_rem_polynom_COEFF 162
bap_replace_initial_polynom_COEFF 155
bap_replace_initial2_polynom_mpz 155
bap_reset_all_settings 134
bap_reset_geobucket_COEFF 166
bap_reset_termanager 136
bap_restart 134
bap_resultant2_Ducos_polynom_COEFF 163
bap_reverse_clot_COEFF 138
bap_reverse_indexed_access 142
bap_reverse_polynom_COEFF 147
bap_scanf_expanded_polynom_COEFF 158
bap_scanf_polynom_COEFF 157
bap_scanf_polynom_eqn_COEFF 158
bap_scanf_product_COEFF 180
bap_scanf_simplify_expanded_polynom_
COEFF 158
bap_scanf_simplify_polynom_COEFF 157
bap_seek_coeff_itercoeff_COEFF 172
bap_seekfirst_itermon_COEFF 174
bap_seeklast_itermon_COEFF 174
bap_separant_and_sepuctum_polynom_mpz 156
bap_separant_polynom_COEFF 156
bap_separant2_polynom_COEFF 156
bap_sepuctum_polynom_mpz 156
bap_set_iterator_indexed_access 144
bap_set_itermon_COEFF 173
bap_set_point_polynom_COEFF 165
bap_set_polynom_COEFF 149
bap_set_polynom_crk_COEFF 149
bap_set_polynom_geobucket_COEFF 167
bap_set_polynom_monom_COEFF 149
bap_set_polynom_numer_denom_mpq 168
bap_set_polynom_one_COEFF 149
bap_set_polynom_term_COEFF 149
bap_set_polynom_variable_COEFF 149
bap_set_polynom_zero_COEFF 149
bap_set_power_COEFF 177
bap_set_power_polynom_COEFF 177
bap_set_product_COEFF 179
bap_set_product_numeric_COEFF 178
bap_set_product_one_COEFF 178
bap_set_product_polynom_COEFF 178
bap_set_product_zero_COEFF 178
bap_set_readonly_polynom_COEFF 149
bap_set_tableof_polynom_COEFF 150
bap_set_tableof_tableof_polynom_COEFF 150
bap_set_tableof_tableof_tableof_polynom_
COEFF 150
bap_set_term_zipterm_termanager 136
bap_set_termstripper 145
bap_set_total_rank_polynom_COEFF 146
bap_set_zipterm_term_termanager 136
bap_set_zipterm_zipterm_termanager 136
bap_signed_numeric_content_polynom_mpz ... 167
bap_simplify_zero_derivatives_of_parameter_
polynom_COEFF 164
bap_sizeof_polynom_COEFF 147
bap_sizeof_tableof_polynom_COEFF 147
bap_sort_clot_COEFF 138
bap_sort_polynom_COEFF 147
bap_sort_product_COEFF 179
bap_sort_tableof_polynom_COEFF 152
bap_split_polynom_COEFF 157
bap_strip_term_termstripper 146
bap_sub_geobucket_COEFF 166
bap_sub_polynom_COEFF 158
bap_submulmon_polynom_COEFF 159
bap_submulrk_polynom_COEFF 159
bap_swap_itermon_clot_COEFF 140
bap_swapindex_iterator_indexed_access 144
bap_switch_ring_polynom_COEFF 147
bap_switch_ring_termanager 137
bap_switch_ring_termstripper 146
bap_term_itercoeff_COEFF 171
bap_term_itermon_clot_COEFF 139
bap_term_itermon_COEFF 173
bap_terminate 134
bap_total_degree_polynom_COEFF 153
bap_total_order_polynom_COEFF 153
bap_uni_Diophante_polynom_mpzm 170
bap_worth_garbage_zipterm_termanager 137
bap_write_all_creator_clot_COEFF 141
bap_write_all_creator_COEFF 175
bap_write_creator_clot_COEFF 140
bap_write_creator_COEFF 175
bap_write_creator_indexed_access 144
bap_write_exquo_all_creator_clot_mpz 141
bap_write_exquo_all_creator_mpz 176
bap_write_mul_all_creator_clot_COEFF 141
bap_write_mul_all_creator_COEFF 176
bap_write_neg_all_creator_clot_COEFF 141
bap_write_neg_all_creator_COEFF 175
bap_write_neg_creator_clot_COEFF 141
bap_write_neg_creator_COEFF 175
bap_write_term_creator_clot_COEFF 141
bas_add_edge_DL_tree 270
bas_add_root_DL_tree 269
bas_compute_beta_task 261
bas_constant_variables_pair 263
bas_Denef_Lipshitz 265
bas_Denef_Lipshitz_aux 264
bas_Denef_Lipshitz_leaf 261
bas_Denef_Lipshitz_resume 265
bas_dot_DL_tree 270

bas_duplicate_mumax_leaf_context 255
bas_get_action_on_task 259
bas_has_a_defining_ODE 255
bas_Hurwitz_coeffs 262
bas_indices_in_regchain_of_all_defining_ODE 255
bas_init_DL_edge 266
bas_init_DL_tree 269
bas_init_DL_vertex 267
bas_init_leaf_context 255
bas_init_pair 263
bas_init_task 259
bas_ith_vertex_DL_tree 270
bas_new_DL_edge 266
bas_new_DL_tree 269
bas_new_DL_vertex 268
bas_new_leaf_context 255
bas_new_pair 263
bas_new_task 259
bas_next_number_DL_tree 269
bas_nonnegative_integer_roots 261
bas_printf_DL_edge 267
bas_printf_DL_tree 270
bas_printf_DL_vertex 268
bas_printf_leaf_context 256
bas_printf_pair 264
bas_printf_task 261
bas_prolongate_binomials_leaf_context 256
bas_prolongate_pair 265
bas_recompute_beta_task 261
bas_reset_all_settings 253
bas_reset_DL_tree 269
bas_reset_DL_vertex 268
bas_restart 253
bas_scanf_pair 264
bas_secure_k_task 260
bas_secure_r_task 260
bas_select_pairs 263
bas_series_coefficients_pair 263
bas_set_all_leaves_entries_leaf_context . 256
bas_set_aykrd_DL_vertex 268
bas_set_aykrd_vertex_DL_tree 270
bas_set_but_change_regchain_and_mumax_task 260
bas_set_context_task_pair 263
bas_set_DL_edge 266
bas_set_DL_tree 269
bas_set_DL_vertex 268
bas_set_leaf_context 255
bas_set_leaf_context_task 260
bas_set_pair 263
bas_set_single_leaf_entries_leaf_context . 256
bas_set_task 260
bas_set_tsd_DL_edge 266
bas_set_vertex_consistency_DL_tree 270
bas_sizeof_pair 264
bas_switch_ring_pair 264
bas_terminate 253
bas_typeof_action_on_task_to_string 259
bas_typeof_DL_edge_to_string 266
bav_add_dictionary_symbol 112
bav_add_dictionary_variable 126
bav_annihilating_derivations_of_parameter 115
bav_block_index_parameter 117
bav_block_index_range_indexed_group 117
bav_block_index_symbol 117
bav_block_sort_tableof_variable 123
bav_cancel_PFE_settings 95
bav_compare_rank 127
bav_compare_stripped_term 133
bav_compare_symbol 110
bav_compare_term 133
bav_constant_rank 127
bav_constant_rank2 127
bav_current_ordering 103
bav_degree_term 130
bav_delete_independent_values_point 133
bav_depends_on_zero_derivatives_of_parameter_term 132
bav_derivation_between_derivatives 120
bav_derivation_index_to_derivation 117
bav_diff_term 132
bav_diff_variable 119
bav_diff2_variable 119
bav_disjoint_term 131
bav_disjoint_variables 119
bav_equal_rank 127
bav_equal_term 130
bav_exquo_term 132
bav_exquo_term_variable 132
bav_gcd_term 131
bav_get_dictionary_symbol 112
bav_get_dictionary_variable 126
bav_get_notations 122
bav_get_settings_common 95
bav_get_settings_ordering 100
bav_get_settings_parameter 114
bav_get_settings_rank 126
bav_get_settings_symbol 109
bav_get_settings_variable 118
bav_get_typed_ident_from_symbol 104
bav_gt_index_variable 123
bav_gt_rank 127
bav_gt_term 130
bav_gt_variable 123
bav_has_varmax_current_ordering 100
bav_independent_variables 124
bav_index_in_rigs_and_subscript_to_variable 121
bav_indexed_string_to_variable 120
bav_init_block 106
bav_init_dictionary_symbol 112
bav_init_dictionary_variable 125
bav_init_differential_ring 96
bav_init_ordering 100

bav_init_parameter 114
bav_init_parameters 114
bav_init_rank 126
bav_init_symbol 109
bav_init_term 128
bav_init_typed_ident 104
bav_int_variable 119
bav_is_a_derivation 110
bav_is_a_parameter 115
bav_is_constant_rank 127
bav_is_constant_variable 118
bav_is_d_derivative 120
bav_is_derivative 120
bav_is_differentially_ambiguous_point 133
bav_is_empty_block 106
bav_is_empty_differential_ring 97
bav_is_factor_term 132
bav_is_one_term 129
bav_is_proper_derivative 120
bav_is_subranking 108
bav_is_subscripted_symbol 110
bav_is_zero_derivative_of_parameter 116
bav_is_zero_rank 127
bav_lcd_variable 119
bav_lcm_tableof_term 131
bav_lcm_term 131
bav_leader_term 130
bav_leading_degree_term 130
bav_leading_rank_term 130
bav_lt_rank 127
bav_lt_term 130
bav_maximal_degree_term 130
bav_mul_term 131
bav_mul_term_rank 131
bav_mul_term_variable 131
bav_new_block 106
bav_new_differential_ring 97
bav_new_ordering 100
bav_new_parameter 114
bav_new_rank 126
bav_new_symbol 109
bav_new_term 128
bav_new_typed_ident 104
bav_new_variable 117
bav_next_derivative 120
bav_not_a_symbol 109
bav_not_a_variable 117
bav_operator_between_derivatives 120
bav_order_variable 118
bav_order_zero_variable 119
bav_pow_term 132
bav_printf_block 106
bav_printf_default_rank 127
bav_printf_default_symbol 111
bav_printf_diff_variable 123
bav_printf_inert_diff_variable 123
bav_printf_jet_variable 122
bav_printf_jet_wesb_variable 122
bav_printf_jet0_variable 122
bav_printf_LaTeX_variable 122
bav_printf_list_rank 128
bav_printf_maple_D_variable 123
bav_printf_numbered_symbol 111
bav_printf_ordering 101
bav_printf_parameter 116
bav_printf_parameter_dependencies 116
bav_printf_python_Derivative_variable 123
bav_printf_rank 128
bav_printf_stars_rank 128
bav_printf_symbol 111
bav_printf_term 133
bav_printf_variable 122
bav_pull_ordering 103
bav_push_ordering 103
bav_R_add_block_to_all_orderings 101
bav_R_ambiguous_symbols 97
bav_R_append_tableof_typed_ident_block ... 105
bav_R_compare_variable 102
bav_R_copy_ordering 102
bav_R_create_differential_ring 97
bav_R_free_ordering 102
bav_R_free_temporary_variable 125
bav_R_lower_block_range_indexed_strings . 103
bav_R_new_derivative 124
bav_R_new_ranking 101
bav_R_new_symbol_and_variable 98
bav_R_new_temporary_variable 124
bav_R_set_block 106
bav_R_set_differential_ring 97
bav_R_set_maximal_variable 124
bav_R_set_minimal_variable 124
bav_R_set_ordering 102
bav_R_set_parameters 114
bav_R_set_parameters_tableof_parameter ... 114
bav_R_set_symbol 111
bav_R_set_tableof_block 106
bav_R_set_tableof_ordering 102
bav_R_set_tableof_typed_ident 105
bav_R_set_tableof_typed_ident_tableof_
block 105
bav_R_set_typed_ident 104
bav_R_set_variable 124
bav_R_sort_tableof_variable 102
bav_R_string_to_existing_derivation 111
bav_R_string_to_existing_symbol 111
bav_R_string_to_existing_variable 124
bav_R_swap_ordering 102
bav_realloc_term 129
bav_reset_all_settings 92
bav_reset_block 106
bav_reset_dictionary_symbol 112
bav_reset_dictionary_variable 125
bav_reset_notations 122
bav_reset_ordering 100
bav_restart 92
bav_restore_PFE_settings 95

bav_scanf_block ... 106
bav_scanf_default_symbol ... 110
bav_scanf_diff_variable ... 121
bav_scanf_inert_diff_variable ... 121
bav_scanf_jet_variable ... 121
bav_scanf_jet0_variable ... 121
bav_scanf_maple_all_variable ... 122
bav_scanf_maple_D_variable ... 122
bav_scanf_ordering ... 101
bav_scanf_parameter ... 116
bav_scanf_python_all_variable ... 122
bav_scanf_python_D_variable ... 121
bav_scanf_python_Derivative_variable ... 121
bav_scanf_rank ... 128
bav_scanf_symbol ... 110
bav_scanf_term ... 133
bav_scanf_variable ... 121
bav_scanf2_symbol ... 111
bav_set_dictionary_symbol ... 112
bav_set_dictionary_variable ... 125
bav_set_ordering ... 101
bav_set_parameter ... 114
bav_set_parameter_with_tableof_string ... 115
bav_set_settings_common ... 95
bav_set_settings_ordering ... 100
bav_set_settings_parameter ... 113
bav_set_settings_rank ... 126
bav_set_settings_symbol ... 109
bav_set_settings_variable ... 118
bav_set_tableof_parameter_with_tableof_string ... 115
bav_set_tableof_tableof_symbol ... 110
bav_set_tableof_term ... 129
bav_set_term ... 129
bav_set_term_one ... 129
bav_set_term_ordering ... 133
bav_set_term_rank ... 129
bav_set_term_tableof_variable ... 132
bav_set_term_variable ... 129
bav_shift_term ... 129
bav_sizeof_block ... 105
bav_sizeof_dictionary_symbol ... 112
bav_sizeof_dictionary_variable ... 125
bav_sizeof_differential_ring ... 97
bav_sizeof_ordering ... 100
bav_sizeof_parameter ... 115
bav_sizeof_parameters ... 114
bav_sizeof_symbol ... 109
bav_sizeof_tableof_block ... 106
bav_sizeof_tableof_ordering ... 100
bav_sizeof_tableof_parameter ... 115
bav_sizeof_tableof_typed_ident ... 104
bav_sizeof_typed_ident ... 104
bav_smallest_greater_variable ... 117
bav_sort_tableof_term ... 131
bav_sort_tableof_variable ... 123
bav_sort_term ... 131
bav_strip_term ... 129
bav_subscript_of_symbol ... 110
bav_switch_ring_symbol ... 110
bav_switch_ring_term ... 128
bav_switch_ring_variable ... 118
bav_symbol_to_variable ... 117
bav_symbol_type_variable ... 119
bav_term_at_point_int_p ... 132
bav_terminate ... 92
bav_total_degree_term ... 130
bav_total_order_term ... 130
bav_total_order_variable ... 118
bav_variable_number ... 124
bav_zero_rank ... 127
baz_add_ratfrac ... 191
baz_add_ratfrac_polynom_mpz ... 188
baz_collect_terms_tableof_ratfrac ... 193
baz_compare_rank_ratfrac ... 190
baz_content_polynom_mpz ... 183
baz_content_tableof_polynom_mpz ... 184
baz_denom_ratfrac ... 189
baz_depend_ratfrac ... 189
baz_derivative_to_subscripted ... 199
baz_diff_ratfrac ... 193
baz_div_ratfrac ... 192
baz_equal_ratfrac ... 189
baz_eval_to_polynom_at_point_int_p_ratfrac ... 193
baz_eval_to_ratfrac_at_point_ratfrac_polynom_mpz ... 186
baz_eval_to_ratfrac_at_point_ratfrac_ratfrac ... 201
baz_evaluate_to_ratfrac_at_point_ratfrac_ratfrac ... 201
baz_extended_Zassenhaus_gcd_tableof_polynom_mpz ... 183
baz_factor_easy_polynom_mpz ... 184
baz_factor_easy_product_mpz ... 185
baz_factor_numeric_content_product_mpz ... 186
baz_factor_polynom_mpq ... 185
baz_factor_polynom_mpz ... 185
baz_gcd_polynom_mpz ... 183
baz_gcd_pquo_polynom_mpz ... 182
baz_gcd_prem_polynom_mpz ... 182
baz_gcd_product_mpz ... 186
baz_gcd_pseudo_division_polynom_mpz ... 182
baz_gcd_tableof_polynom_mpz ... 183
baz_gcd_univariate_tableof_polynom_mpz ... 182
baz_gcdheu_tableof_polynom_mpz ... 183
baz_genpoly_polynom_mpz ... 182
baz_get_settings_prolongation_pattern ... 196
baz_init_prolongation_pattern ... 196
baz_init_ratfrac ... 187
baz_init_readonly_ratfrac ... 187
baz_init_rel_ratfrac ... 195
baz_init_series_mapping ... 198
baz_init_value_ratfrac ... 200
baz_initial_ratfrac ... 190
baz_invert_ratfrac ... 192

baz_is_constant_ratfrac ... 189
baz_is_numeric_ratfrac ... 189
baz_is_one_ratfrac ... 189
baz_is_zero_ratfrac ... 189
baz_lcoeff_and_reductum_ratfrac ... 190
baz_leader_ratfrac ... 190
baz_mark_indets_ratfrac ... 190
baz_monomial_reduce_polynom_mpz ... 185
baz_mul_ratfrac ... 192
baz_mul_ratfrac_numeric ... 192
baz_mul_ratfrac_numeric_mpq ... 192
baz_mul_ratfrac_polynom_mpq ... 189
baz_mul_ratfrac_polynom_mpz ... 188
baz_mul_ratfrac_variable ... 192
baz_neg_ratfrac ... 191
baz_new_prolongation_pattern ... 197
baz_new_ratfrac ... 187
baz_new_readonly_ratfrac ... 187
baz_new_rel_ratfrac ... 195
baz_new_series_mapping ... 198
baz_new_value_ratfrac ... 200
baz_normalize_numeric_initial_ratfrac ... 191
baz_numer_ratfrac ... 189
baz_physort_ratfrac ... 191
baz_positive_integer_roots_polynom_mpq ... 187
baz_positive_integer_roots_polynom_mpz ... 187
baz_positive_roots_polynom_mpq ... 186
baz_pow_ratfrac ... 192
baz_primpart_polynom_mpz ... 184
baz_printf_prolongation_pattern ... 198
baz_printf_ratfrac ... 195
baz_printf_rel_ratfrac ... 196
baz_prolongate_point_ratfrac_term ... 200
baz_prolongate_point_ratfrac_using_pattern_term ... 201
baz_prolongate_point_ratfrac_using_pattern_variable ... 200
baz_prolongate_point_ratfrac_variable ... 200
baz_rank_ratfrac ... 190
baz_rat_bilge_mpz ... 193
baz_reduce_numeric_ratfrac ... 191
baz_reduce_ratfrac ... 191
baz_reductum_ratfrac ... 190
baz_reset_all_settings ... 181
baz_reset_prolongation_pattern ... 197
baz_restart ... 181
baz_scanf_expanded_ratfrac ... 194
baz_scanf_prolongation_pattern ... 197
baz_scanf_ratfrac ... 194
baz_scanf_rel_ratfrac ... 195
baz_scanf_simplify_expanded_ratfrac ... 194
baz_scanf_simplify_ratfrac ... 194
baz_separant_ratfrac ... 192
baz_separant2_ratfrac ... 192
baz_set_point_ratfrac ... 200
baz_set_prolongation_pattern ... 197
baz_set_ratfrac ... 188
baz_set_ratfrac_fraction ... 189
baz_set_ratfrac_one ... 188
baz_set_ratfrac_polynom_mpz ... 188
baz_set_ratfrac_rel_ratfrac ... 195
baz_set_ratfrac_term ... 188
baz_set_ratfrac_zero ... 188
baz_set_rel_ratfrac ... 195
baz_set_series_mapping ... 199
baz_set_settings_prolongation_pattern ... 196
baz_set_spx_series_mapping ... 199
baz_set_tableof_ratfrac ... 188
baz_set_tableof_tableof_ratfrac ... 188
baz_set_tableof_tableof_tableof_ratfrac ... 188
baz_set_value_ratfrac ... 200
baz_sizeof_prolongation_pattern ... 197
baz_sizeof_ratfrac ... 193
baz_sizeof_tableof_ratfrac ... 193
baz_sizeof_tableof_tableof_ratfrac ... 194
baz_sort_ratfrac ... 191
baz_squarefree_polynom_mpz ... 184
baz_sub_ratfrac ... 191
baz_subscripted_to_derivative ... 199
baz_switch_ring_prolongation_pattern ... 197
baz_switch_ring_ratfrac ... 194
baz_terminate ... 182
baz_variable_mapping_prolongation_pattern ... 197
baz_yet_another_point_int_p_mpz ... 185
baz_Yun_polynom_mpz ... 184

Type Index

ba0_analex_token_fifo 47
ba0_atomic_format_component 31
ba0_binary_operation 51
ba0_cmp_function 51
ba0_cmp2_function 51
ba0_composite_format_component 31
ba0_copy_function 32
ba0_dictionary 86
ba0_dictionary_int_p 87
ba0_dictionary_string 88
ba0_dictionary_typed_string 90
ba0_exception 39
ba0_exception_code 39
ba0_format 31
ba0_format_component 31
ba0_garbage_code 29
ba0_garbage1_function 29
ba0_garbage2_function 29
ba0_gc_info 29
ba0_global 13
ba0_indexed_string 79
ba0_indexed_string_indices 80
ba0_init_function 51
ba0_initialized_global 11
ba0_input_device 44
ba0_interval_mpq 73
ba0_list 51
ba0_listof_double 53
ba0_listof_int_p 52
ba0_mark 21
ba0_matrix 59
ba0_matrixof_double 61
ba0_matrixof_int_p 60
ba0_new_function 51
ba0_output_device 42
ba0_PFE_settings 18
ba0_point 63
ba0_point_double 66
ba0_point_int_p 65
ba0_printf_function 32
ba0_range_indexed_group 82
ba0_scanf_function 32
ba0_stack 21
ba0_table 53
ba0_tableof_double 57
ba0_tableof_indexed_string_indices 80
ba0_tableof_int_p 56
ba0_ternary_operation 51
ba0_token 47
ba0_typeof_device 42
ba0_typeof_format_component 31
ba0_typeof_interval 72
ba0_typeof_matrix_memory_layout 59
ba0_typeof_token 46
ba0_unary_function 50
ba0_unary_operation 51
ba0_unary_predicate 51
ba0_value 62
ba0_value_double 63
ba0_value_int_p 62
bad_activity_level_splitting_tree 235
bad_attchain 204
bad_base_field 220
bad_critical_pair 232
bad_global 203
bad_initialized_global 203
bad_intersectof_regchain 213
bad_preparation_equation 251
bad_preparation_term 250
bad_property_attchain 204
bad_quadruple 244
bad_quench_map 216
bad_regchain 206
bad_selection_strategy 234
bad_shapeof_splitting_vertex 238
bad_splitting_control 242
bad_splitting_edge 240
bad_splitting_tree 235
bad_splitting_vertex 238
bad_typeof_critical_pair 231
bad_typeof_derivative_to_reduce 224
bad_typeof_dimension_lower_bound 242
bad_typeof_Euclid 227
bad_typeof_inclusion_test_result 211
bad_typeof_reduction 224
bad_typeof_reduction_strategy 223
bad_typeof_redzero_strategy 223
bad_typeof_regularize_strategy 226
bad_typeof_selection_strategy 234
bad_typeof_splitting_edge 240
bai_coefficient 275
bai_external_C_function 271
bai_radau5_dense_output_state 281
bai_radau5_idid 285
bai_radau5_initial_values 278
bai_radau5_input 279
bai_radau5_integration_state 280
bai_radau5_recursive_polynomial_data 277
bai_radau5_restore_point 282
bai_radau5_result 279
bai_radau5_session 284
bai_radau5_solution 285
bai_radau5_tolerances 278
bai_radau5_typeof_tolerance 278
bai_recursive_polynomial 276
bai_typeof_variable 271
bai_variable_bindings 272
bap_clot_COEFF 137

bap_composite_number 138
bap_creator_COEFF 174
bap_creator_indexed_access 144
bap_geobucket_COEFF 166
bap_indexed_access 142
bap_itermon_clot_COEFF 138
bap_itermon_COEFF 172
bap_polynom_COEFF 146
bap_power_COEFF 176
bap_product_COEFF 176
bap_rank_code 173
bap_sequential_access 142
bap_table2of_monom_COEFF 137
bap_tableof_table2of_monom_COEFF 137
bap_termanager 135
bap_termstripper 145
bap_typeof_total_rank 174
bas_activity_level_DL_tree 268
bas_DL_edge 266
bas_DL_tree 268
bas_DL_vertex 267
bas_leaf_context 254
bas_pair 262
bas_task 257
bas_typeof_action_on_task 259
bas_typeof_consistency_vertex 267
bas_typeof_DL_edge 265
bav_block 105
bav_dictionary_symbol 111
bav_dictionary_variable 125
bav_differential_ring 95
bav_global 94
bav_initialized_global 93
bav_ordering 98
bav_parameter 113
bav_parameters 113
bav_PFE_settings 95
bav_rank 126
bav_subranking 107
bav_symbol 108
bav_tableof_ordering 99
bav_tableof_parameter 113
bav_term 128
bav_typed_ident 103
bav_typeof_ident 103
bav_typeof_symbol 108
bav_variable 116
baz_initialized_global 181
baz_point_ratfrac 199
baz_prolongation_pattern 196
baz_ratfrac 187
baz_rel_ratfrac 195
baz_series_mapping 198
baz_typeof_realroot_interval 186
baz_typeof_relop 195
baz_value_ratfrac 199