CoCoALib 0.99 documentation

John Abbott and Anna Bigatti

2011,2013

Contents

1 INSTALL (John Abbott and Anna Bigatti)

1.1 INSTALLATION guide for CoCoALib

CoCoALib is supplied as **SOURCE** code in C++, and so must be **COMPILED** before you can use it – instructions on how to do this are below.

1.1.1 Prerequisites

Before compilation you must ensure that you have available:-

- the GNU `make` program (other versions may work too);
- a C++ compiler together with the standard C++ libraries (*e.g.* `g++`)
- an installation of GMP (version 4.2.1 or later) – see `http://gmplib.org/`
- if you want to build `CoCoA-5` too, you need the **BOOST libraries** (see `http://www.boost.org/`); more details are in [[src/CoCoA-5/INSTALL]]

1.1.2 Compilation of CoCoALib

Use the `cd` comand to go to the root directory `CoCoALib-nnn`. In most cases the following two commands will suffice:

```
./configure
make
```

The command `make` compiles CoCoALib (and puts it in `lib/libcocoa.a`); it also compiles & runs the test suite, and will compile CoCoA-5 if possible. The compilation generally takes a few minutes. If there were no problems you'll get the reassuring message:

```
Good news: all tests passed
```

Notes

(1) The configure script looks for the GMP library, and makes a few checks. It assumes your compiler is `g++`. If it encounters a problem, it will print out a helpful error message telling you.

(2) The command `make library` will compile the library but not run the tests. The command `make check` will run the tests – they are in `src/tests/`.

(3) For the adventurous: the command

```
./configure --help
```

explains the various options the script recognizes. Also look at `INSTALL` advanced (Sec.1)

1.1.3 Documentation & Examples

Main documentation for CoCoALib: `index` (Sec.**??**)

Example programs using CoCoALib: [[../../examples/index]]

1.1.4 Microsoft Windows

If you have Microsoft Windows, read the file `INSTALL` MicrosoftWindows (Sec.1)

1.1.5 In Case of Trouble

If you encounter problems while using CoCoALib (or trying to compile it), the best way to let us know is to report the issue via

`http://cocoa.dima.unige.it/redmine/`

Please tell us also the platform and compiler you are using.

Alternatively you can send us email at `cocoa@dima.unige.it`

2 INSTALL-advanced (John Abbott and Anna Bigatti)

2.1 Advanced Installation Options

By default `CoCoALib` allows quite polynomials of quite high degree (*e.g.* typically beyond 1000000). If you are sure that degrees will remain small (*e.g.* below 1000) then you *might obtain better performance* by editing the source file `include/CoCoA/config.H` so that the `typedef` for `SmallExponent_t` is `unsigned short` instead of `unsigned int`. But **beware** that CoCoALib does not generally check for exponent overflow during polynomial arithmetic!

3 INSTALL-MicrosoftWindows (John Abbott and Anna Bigatti)

3.1 Guidelines for installing CoCoA on a Microsoft Windows computer

You can build `CoCoALib` and `CoCoA-5` on a *Microsoft Windows* computer by using `Cygwin`, a free package which provides a Linux-like environment (see `http://www.cygwin.com/`).

Once you have installed Cygwin, start its terminal emulator, and then follow the usual instructions for compiling CoCoA.

3.1.1 Installing Cygwin

WARNING: installing Cygwin can take quite some time

Download the script `setup.exe` from the Cygwin website. Start the script and choose *install from internet*. Using that script select the following extension packages:

- gcc-g++
- make
- m4
- libboost-devel
- libboost-1.48
- libgmp-devel
- emacs

If you want to build the "CoCoA-5' GUI, you must obtain also these extension packages for Cygwin

- qt4-devel-tools
- libqtcore4
- libqtcore4devel
- libqtgui4
- libqtgui4-devel
- libqtxml4-devel

- xorg-server
- xinit
- emacs-X11 (not necessary, but probably helpful)

3.1.2 In Case of Trouble

We cannot really help you, as we have almost no experience ourselves. Try searching on the internet...

4 INTRODUCTION (John Abbott)

4.1 Quick Summary: CoCoALib and CoCoA-5

CoCoA-5 is an easy-to-use interactive system for computations in commutative algebra; it contains an on-line manual accessible via the `?` command.

CoCoALib is a C++ *library* of functions for computations in commutative algebra.

This introduction is part of the documentation for *CoCoALib*; to use the library you will need some *basic knowledge* of the C++ programming language.

4.2 Getting Started

The first step is to compile the software: see `INSTALL` (Sec.1)

4.2.1 Using CoCoALib

As we know that no one likes to read documentation, the best place to start is by looking at the examples/ directory full of sample code using CoCoALib.

Writing Your Own Programs

The simplest approach is to copy the example program `ex-empty.C` and modify that (see guide).

If you want to experiment with CoCoALib using a different directory, just copy `examples/Makefile` into your directory and change the line

```
COCOA_ROOT=...
```

so that it specifies the full path of `CoCoALib-XX`, for instance

```
COCOA_ROOT=/Users/bigatti/CoCoALib-0.99
```

In any case, it is best to start with a copy of `ex-empty.C`.

Debugging with CoCoALib

CoCoALib does offer some help in tracking down bugs in programs which use it. If the preprocessor symbol `CoCoA_DEBUG` is set then various run-time assertions are enabled which perform extra checks in various functions. If you use the compiler `g++` then the simplest way to activate debugging is to modify two lines in the file `configuration/autoconf.mk` – the file contains comments to guide you. You may like to read [assert.html] to learn about `CoCoA_ASSERT`.

4.2.2 Various Forms of Documentation

CoCoALib comes with a collection of hand-written descriptions of its capabilities as well as a collection of example programs showing how to use many of the features of the library. The hope is that the example programs (plus perhaps a little intelligent guesswork) will suffice to answer most questions about CoCoALib. The hand-written

documentation is intended to be more thorough: so less guesswork is needed, but you may have to plough through lots of tedious text to find the detail you're looking for.

The hand-written documentation is split into many files: generally there is one file of documentation for each implementation file in the source code. Furthermore, each file comprises three sections:

- **User Documentation** gives the information a normal user of the library may need to know, principally the function interfaces offered
- **Maintainer Documentation** contains notes and details about how the various functions are implemented in the library, essentially all information that might be needed to comprehend and maintain the code in the future
- **Shortcomings, etc** contains sundry notes about the implementation, for instance ideas on how the implementation might be improved in the future, doubts about certain design choices, and generally any thoughts to be taken into consideration for future versions.

This documentation is in the CoCoALib directory `doc/txt/`, and converted into html (`doc/html/`) and LaTeX (`doc/tex/`) using `txt2tags`.

A template file fo adding to this documentation and some basic instructions for `txt2tags` are in the file doc/txt/empty.txt.

There is also some automatically generated DOXYGEN documentation in [../doxygen/index.html]

We believe that many simple questions are probably best answered by looking at the example programs (and perhaps applying a little intelligent guesswork). The hand-written documentation in the directory doc/ is supposed to be exhaustive (and is doubtless also rather exhausting). The Doxygen files will most likely be of use to those already experienced in using CoCoALib.

4.3 Sundry Important Points

We have tried to give CoCoALib a *natural* interface, but this has not always been possible. Here are the main problem areas:

Powering and Exponentiation

The use of the hat symbol (^) to denote exponentiation is very widespread. **CoCoALib does not allow this** you must use the function `power` instead.

Why not? Because it would be too easy to write misleading code, *i.e.* valid code which does not compute what you would expect. Here is a simple example: `3*x^2` is interpreted by the compiler as `(3*x)^2`. Unfortunately there is no way to make the C++ compiler use the expected interpretation.

Integers and Rationals

The C++ language is not designed to compute directly with unlimited integers or with exact rational numbers; special types (namely `BigInt` (Sec.8) and `BigRat` (Sec.9)) to handle these sorts of values have been added as part of CoCoALib (with the real work being done by the GMP library). Nevertheless the user has to be wary of several pitfalls where code which looks correct at first glance does not produce the right answer.

- rationals must be constructed explicitly, *e.g.* the expression `2/3` is valid C++ but is interpreted as an integer division giving result `0`; instead the rational must be constructed like this `BigRat(2,3)`.
- large integer constants must be converted from a string representation, *e.g.* `n = 99...99;` (with 99 nines) will probably not even compile because of an error about "integer constant too big"; instead such a large value must be handled directly by CoCoALib in a call like `convert(n, "99...99");` where the variable `n` has already been declared to be of type `BigInt` (Sec.8) or `BigRat` (Sec.9).
- the compiler believes it knows how to perform arithmetic between machine integers, but the spectre of overflow continues to haunt such computations. Overflow cannot occur with values of type `BigInt` (Sec.8) but the computations will be much slower than with machine integers. If you are quite sure that large values can never occur then it is fine to use machine integers; otherwise use unlimited integers.

- (AMB: add examples from talk in Kassel)

=== Reporting CoCoALib Bugs and other problems ==

Please let us know if you find any bugs in CoCoALib. Ideally your bug report should include a **small** program which exhibits the bad behaviour with a clear indication of what you think the program should do, and where it apparently goes wrong. The best way to inform us of the problem is to report an *issue* on

```
http://cocoa.dima.unige.it/redmine/
```

If you'd rather not use redmine Forum, you can send email to:

```
cocoa@dima.unige.it
```

5 Coding Conventions (John Abbott)

5.1 User and contributor documentation

This page summarises the coding conventions used in CoCoALib. This document is useful primarily to contributors, but some users may find it handy too. As the name suggests, these are merely guidelines; they are not hard and fast rules. Nevertheless, you should violate these guidelines only if you have genuinely good cause. We would also be happy to receive notification about parts of CoCoALib which do not adhere to the guidelines.

We expect these guidelines to evolve slowly with time as experience grows.

Before presenting the guidelines I mention some useful books. The first is practically a *sine qua non* for the C++ library: **The C++ Standard Library** by Josuttis which contains essential documentation for the C++ library. Unless you already have quite a lot of experience in C++, you should read the excellent books by Scott Meyers: **Effective C++** (the new version), and **Effective STL**. Another book offering useful guidance is **C++ Coding Standards** by Alexandrescu and Sutter; it is a good starting point for setting coding standards.

5.1.1 Names of CoCoA types, functions, variables

All code and "global" variables must be inside the namespace `CoCoA` (or in an anonymous namespace); the only exception is code which is not regarded as an integral part of CoCoA (e.g. the C++ interface to the GMP big integer package).

There are numerous conventions for how to name classes/types, functions, variables, and other identifiers appearing in a large package. It is important to establish a convention and apply it rigorously (plus some common sense); doing so will facilitate maintenance, development and use of the code. (The first three rules follow the convention implicit in **NTL**)

- single word names are all lower-case (*e.g.* `ring`);
- multiple word names have the first letter of each word capitalized, and the words are juxtaposed (rather than separated by underscore, (*e.g.* `PolyRing`);
- acronyms should be all upper-case (*e.g.* `PPM`);
- names of functions returning a boolean start with `Is` (`Are` if argument is a list/vector);
- names of functions returning a `bool3` (Sec.10) start with `Is` and end with `3` (`Are` if argument is a list/vector);
- variables of type (or functions returning a) pointer end with `Ptr`
- data members' names start with `my` (or `Iam/Ihave` if they are boolean);
- a class static member has a name beginning with `our`;
- enums are called `BlahMarker` if they have a single value (*e.g.* `BigInt::CopyFromMPZMarker`) and `BlahFlag` if they have more;
- abbreviations should be used consistently (see below);

- **Abstract base classes** and **derived abstract classes** normally have names ending in `Base`; in contrast, a **derived concrete class** normally has a name ending in `Impl`. Constructors for abstract classes should probably be `protected` rather than `public`.

It is best to choose a name for your function which differs from the names of functions in the C++ standard library, otherwise it can become necessary to use fully qualified names (e.g. `std::set` and `CoCoA::set`) which is terribly tedious. (Personally, I think this is a C++ design fault)

If you are overloading a C++ operator then write the keyword `operator` attached to the operator symbol (with no intervening space). See `ring.H` for some examples.

5.1.2 Order in function arguments

When a function has more than one argument we follow the first applicable of the following rules:

1. the non-const references are the first args, e.g.
   - `myAdd(a,b,c)` as in *a=b+c*,
   - `IsIndetPosPower(long& index, BigInt& exp, pp)`
2. the ring/PPMonoid is the first arg, e.g.
   - `ideal(ring, vector<RingElem>)`
3. the *main actor* is the first arg and the *with respect to* args follow, e.g.
   - `deriv(f, x)`
4. optional args go last, e.g.
   - `NewPolyRing(CoeffRing, NumIndets)`,
   - `NewPolyRing(CoeffRing, NumIndets, ordering)`
5. the arguments follow the order of the common use or *sentence*, e.g.
   - `div(a,b)` for *a/b*,
   - `IndetPower(P, long i, long/BigInt exp)` for *x[i]^exp*,
   - `IsDivisible(a,b)` for *a is divisible by b*,
   - `IsContained(a,b)` for *a is contained in b*
6. strongly related functions behave as if they were overloading (–> optional args go last), (??? is this ever used apart from `NewMatrixOrdering(long NumIndets, long GradingDim, ConstMatrixView OrderMatrix);`???)
7. the more structured objects go first, e.g. ... (??? is this ever used ???)

IMPORTANT we are trying to define a **good set of few rules** which is easy to apply and, above all, respects *common sense*. If you meet a function in CoCoALib not following these rules let us know: we will fix it, or fix the rules, or call it an interesting exception ;-)

Explanation notes, exceptions, and more examples

- We don't think we have any function with 1 and 2 colliding
- The *main actor* is the object which is going to be worked on to get the returning value (usually of the same type), the *with respect to* are strictly constant, e.g.
  - `deriv(f, x)`
  - `NF(poly, ideal)`
- Rule 1 wins on rule 4, e.g.
  - `IsIndetPosPower(index, exp, pp)` and `IsIndetPosPower(pp)`
- Rule 2 wins on rule 4, e.g.
  - `ideal(gens)` and `ideal(ring, gens)`
- we should probably change:
  - `NewMatrixOrdering(NumIndets, GradingDim, M)` into `NewMatrixOrdering(M, GradingDim)`

5.1.3 Abbreviations

The overall idea is that if a given concept in a class or function name always has the same name: either always the full name, or always the same abbreviation. Moreover a given abbreviation should have a unique meaning.

Here is a list for common abbreviations

- `col` – column
- `ctor` – constructor
- `deg` – degree (exceptions: `degree` in class names)
- `div` – divide
- `dim` – dimension
- `elem` – element
- `mat` – matrix (exceptions: `matrix` in class names)
- `mul` – multiply
- `pos` – positive (or should it be `positive`? what about `IsPositive(BigInt N)`?)

Here is a list of names that are written in full

- `assign`
- `one` – not `1`
- `zero` – not `0`

5.2 Contributor documentation

5.2.1 Guidelines from Alexandrescu and Sutter

Here I paraphrase some of the suggestions from their book, picking out the ones I think are less obvious and are most likely to be relevant to CoCoALib.

- Write correct, clean and simple code at first; optimize later.
- Keep track of object ownership; document any "unusual" behaviour.
- Keep implementation details hidden (*e.g.* make data members `private`)
- Use `const` as much as you reasonably can.
- Use prefix ++ and – (unless you specifically do want the postfix behaviour)
- Each class should have a *single* clearly defined purpose; keep it simple!
- Guideline: member fns should be either `virtual` or `public` not both.
- Exception cleanliness: dtors, deallocate and `swap` should never throw.
- Use `explicit` to avoid making unintentional "implicit type conversions"
- Avoid `using` in header files.
- Use `CoCoA_ERROR` for sanity checks on args to public fns, and `CoCoA_ASSERT` for internal fns.
- Use `std::vector` unless some other container is obviously better.
- Avoid casting; if you must, use a C++ style cast (*e.g.* `static_cast`)

5.2.2 Use of "#define"

Excluding the *read once trick* for header files, `#define` should be avoided (even in experimental code). C++ is rich enough that normally there is a cleaner alternative to a `#define`: for instance, `inline` functions, a `static const` object, or a `typedef` – in any case, one should avoid polluting the global namespace.

If you must define a preprocessor symbol, its name should begin with the prefix `CoCoA_`, and the remaining letters should all be capital.

5.2.3 Header Files

The *read once trick* uses preprocessor symbols starting with `CoCoA_` and then finishing with the file name (retaining the capitalization of the file name but with slashes replaced by underscores). The include path passed to the compiler specifies the directory above the one containing the CoCoALib header files, so to include one of the CoCoALib header files you must prefix `CoCoA/` to the name of the file – this avoids problems of ambiguity which could arise if two includable files have the same name. This idea was inspired by NTL.

Include only the header files you really need – this is trickier to determine than you might imagine. The reasons for minimising includes are two-fold: to speed compilation, and to indicate to the reader which external concepts you genuinely need. In header files it often suffices simply to write a forward declaration of a class instead of including the header file defining that class. In implementation files the definition you want may already be included indirectly; in such cases it is enough to write a comment about the indirectly included definitions you will be using.

In header files I add a commented out `using` command immediately after including a system header to say which symbols are actually used in the header file. In the implementation file I write a `using` command for each system symbol used in the file; these commands appear right after the `#include` directive which imported the symbol.

5.2.4 Curly brackets and indentation

Sutter claims curly bracket positioning doesn't matter: he's wrong! Matching curly brackets should be either vertically or horizontally aligned. Indentation should be small (*e.g.* two positions for each level of nesting); have a look at code already in CoCoALib to see the preferred style. Avoid using tabs for indentation as these do not have a universal interpretation.

The `else` keyword indents the same as its matching `if`.

5.2.5 Inline Functions

Use `inline` sparingly. `inline` is useful in two circumstances: for a short function which is called very many times (at least several million), or for an extremely short function (e.g. a field accessor in a class). The first case may make the program faster; the second may make it shorter. You can use a profiler (*e.g.* `gprof`) to count how often a function is called.

There are two potential disadvantages to `inline` functions: they may force implementation details to be publicly visible, and they may cause code bloat.

5.2.6 Exception Safety

Exception Safety is an expression invented/promulgated by Sutter to mean that a procedure behaves well when an exception is thrown during its execution. All the main functions and procedures in CoCoALib should be fully exception safe: either they complete their computations and return normally, or they leave all arguments essentially unchanged, and return exceptionally. A more relaxed approach is acceptable for functions/procedures which a normal library user would not call directly (*e.g.* non-public member functions): it suffices that no memory is leaked (or other resources lost). Code which is not fully exception-safe should be clearly marked as such.

Consult one of Sutter's (irritating) books for more details.

5.2.7 Dumb/Raw Pointers

If you're using dumb/raw pointers, improve your design!

Dumb/raw pointers should be used only as a last resort; prefer C++ references or `std::auto_ptr<...>` if you can. Note that it is especially hard writing exception safe code which contains dumb/raw pointers.

5.2.8 Preprocessor Symbols for Controlling Debugging

During development it will be useful to have functions perform *sanity checks* on their arguments. For general use, these checks could readily produce a significant performance hit.

Compilation without setting any preprocessor variables should produce fast code (i.e. without non-vital checks). Instead there is a preprocessor symbol (`CoCoA_DEBUG`) which can be set to turn on extra sanity checks. Currently if `CoCoA_DEBUG` has value zero, all non-vital checks are disabled; any non-zero value enables all additional checks.

There is a macro `CoCoA_ASSERT(...)` which will check that its argument yields `true` when `CoCoA_DEBUG` is set; if `CoCoA_DEBUG` is not set it does nothing (not even evaluating its argument). This macro is useful for conducting extra sanity checks during debugging; it should **not be used** for checks that must always be performed (*e.g.* in the final optimized compilation).

There is currently no official preprocessor symbol for (de)activating the gathering of statistics.

NB I wish to avoid having a plethora of symbols for switching debugging on and off in different sections of the code, though I do accept that we may need more than just one or two symbols.

5.2.9 Errors and Exceptions

During development

Conditions we want to verify **solely during development** (i.e. when compiling with `-DCoCoA_DEBUG`) can be checked simply by using the macro `CoCoA_ASSERT` with argument the condition. Should the condition be false, a `CoCoA::ErrorInfo` object is thrown – this will cause an abort if not caught. The error message indicates the file and line number of the failing assertion. If the compilation option `-DCoCoA_DEBUG` is not enabled then the macro does nothing whatsoever. An example of its use is:

```
CoCoA_ASSERT(index <= 0 && index < length);
```

Always

A different mechanism is to be used for conditions which must be checked even **after development is completed**.

What should happen when one tries to divide by zero? Or even asks for an exact division between elements that do not have an exact quotient (in the given ring)?

Answer: call the macro `CoCoA_ERROR(err_type, location)` where `err_type` should be one of the error codes listed in `error.H` and `location` is a string saying where the error was detected (e.g. the name of the function which discovered it). Here is an example

```
if (trouble)
  CoCoA_ERROR(ERR::DivByZero, "applying partial ring homomorphism");
```

The macro `CoCoA_ERROR` never returns: it will throw a `CoCoA::ErrorInfo` object. See the example programs for the recommended way of catching and handling exceptions: so that an informative message can be printed out. See `error.txt` for advice on debugging when an unexpected CoCoA error is thrown.

5.2.10 Functions Returning Complex Values

C++ tends to copy the return value of a function; this is undesirable if the value is potentially large and complex. An obvious alternative is to supply as argument a reference into which the result will be placed. If you choose to return the value via a reference argument then make the reference argument **the first one**.

```
myAdd(rawlhs, rawx, rawy);  // stands for:  lhs = x + y
```

5.2.11 Spacing and Operators

All **binary operators** should have one space before and one space after the operator name (unless both arguments are particularly short and simple). **Unary operators** should not be separated from their arguments by any spaces. Avoid spaces between **function** names and the immediately following bracket.

```
expr1 + expr2;
!expr;
UsefulFunction(args);
```

6 ApproxPts (John Abbott, Anna M. Bigatti)

6.1 Examples

- ex-ApproxPts1.C

6.2 User documentation

`ApproxPts` offers three functions for preprocessing sets of approximate points whose coordinates are given as values of type `double`. Given a large set of approximate points with considerable overlap of the error boxes of adjacent points, the preprocessing algorithms determine a smaller set of approximate points which preserve the geometrical disposition of the original points but with little or no overlap of the error boxes. In general, the output points **do not form a subset** of the original points.

Details of the underlying algorithms are in the article **Thinning Out Redundant Empirical Data** by Abbott, Fassino, Torrente, and published in *Mathematics in Computer Science* (vol. 1, no. 2, pp. 375-392, year 2007). For a fully detailed description of the methods and the context in which they were developed refer to **Laura Torrente's PhD thesis**: (*Applications of Algebra in the Oil Industry*, Scuola Normale Superiore di Pisa, 2009). The thesis is available at the URL Laura's thesis

6.2.1 Operations

Here is a quick summary of the functions.

```
typedef ApproxPts::PointR ApproxPt;  // actually std::vector<RingElem>
vector<ApproxPt>  OriginalPoints;    // the coords of the original approx pts
vector<RingElem>  epsilon;           // epsilon[i] is semiwidth of error box in dimension i
vector<ApproxPt>  NewPoints;         // will be filled with the preprocessed points
vector<long>      weights;           // will be filled with the weights of the representatives

PreprocessPts(NewPoints, weights, OriginalPoints, epsilon);
PreprocessPtsGrid(NewPoints, weights, OriginalPoints, epsilon);
PreprocessPtsAggr(NewPoints, weights, OriginalPoints, epsilon);
PreprocessPtsSubdiv(NewPoints, weights, OriginalPoints, epsilon);
```

All the algorithms work by partitioning the original points into subsets, and then choosing the average of each subset as the representative of those original points. The **weight** of each representative is just the number of original points in the corresponding partition. The algorithms offer differing trade-offs between speed and number of representatives.

`PreprocessPtsGrid` **This algorithm is the fastest but the results tend to be rather crude; it is possible that some**

`PreprocessPtsAggr` **This algorithm gives much better results than** `PreprocessPtsGrid` **but can take considerably lo**

`PreprocessPtsSubdiv` **This algorithm generally gives the best results (*i.e.* fewest output points, and best visual di**

`PreprocessPts` **makes a (not very) intelligent choice between** `PreprocessPtsAggr` **and** `PreprocessPtsSubdiv` **aiming t**

6.3 Maintainer documentation for files ApproxPts.H and ApproxPts.C

All the preprocessing algorithms rescale their inputs so that the error widths in each dimension are all equal to 1. The main work is done with these rescaled points, and at the very end the results are scaled back.

`PreprocessPtsGrid` might be better if we were to use `std::maps`, but it seems fast enough as is. From the theory, each input point is associated to a unique grid point; `GridNearPoint` effects this association. We build up a table of useful grid points by considering each input point in turn: if the associated grid point is already in our table of grid points, we simply append the new input point to the grid point's list of associated original points, otherwise we add the new grid point to the table and place the input point as the first element in its list of associated original points. Finally we compute the averages of each list of original points associated to a fixed grid point. These averages are our result along with the cardinalities of the corresponding list.

`PreprocessPtsAggr` implements an **aggregative algorithm**: initially the original points are split into subsets each containing exactly one original point, then iteratively nearby subsets are coalesced into larger subsets provided each original point of the two subsets is not too far from the "centre of gravity" of the coalesced set – this proviso is necessary as otherwise there are pathological examples.

`PreprocessPtsSubdiv` implements a **subdivision algorithm**. Initially all original points are placed into a single partition. Then iteratively we seek the original point furthest from the average of its subset. If this distance is below the threshold then we stop (all original points are sufficiently well represented by the averages of their subsets). Otherwise we separate the worst represented original point into a new subset initially containing just itself. Now we redistribute the original points: we do this by minimizing the sum of the squares of the L2 distances of the original points from their respective representatives.

6.4 Bugs, Shortcomings and other ideas

I do not like the typedef for `ApproxPts::ApproxPt` because the name seems very redundant; I am also uneasy about having a `typedef` in a header file – perhaps it should be a genuine class?

The preprocessing algorithms should really receive input as a pair of iterators, and the output should be sent to an output iterator. But such an interface would rather uglify the code – what to do???

7 assert (John Abbott)

7.1 Examples

- ex-PolyIterator2.C

7.2 User documentation for files assert.H and assert.C

The only part of `assert.H` which a normal CoCoALib user might find useful is the `CoCoA_ASSERT` macro: it is intended to be a debugging aid.

The `CoCoA_ASSERT` macro does absolutely nothing (not even evaluating its argument) unless the compilation flag `CoCoA_DEBUG` is set. If that flag is set then the macro evaluates its argument to a boolean result which is then tested: if the result is true nothing further happens, if the result is false then the function `CoCoA::AssertionFailed` is called with some arguments indicating which `CoCoA_ASSERT` macro call obtained the false value. The `AssertionFailed` function prints out an informative message on `std::cerr` and then throws a `CoCoA::ERR::AssertFail` exception.

The file `assert.H` contains the following: - definition of the `CoCoA_ASSERT` macro to aid debugging, and the related function `AssertionFailed`

7.2.1 Debugging

During debugging, a debugger can be used to intercept calls to the function `CoCoA::AssertionFailed` which will stop the program just before throwing the `CoCoA::ERR::AssertFail` exception. This should enable one to find more easily the cause of the problem.

For example, in **gdb** type

```
break CoCoA::AssertionFailed
```

and then go `up` (perhaps repeatedly) to the offending line.

7.3 Maintainer documentation for files assert.H and assert.C

The macro name `CoCoA_ASSERT` is rather cumbersome, but must contain the prefix `CoCoA_` since macro names cannot be placed in C++ namespaces. The two definitions of the macro (debugging and non-debugging cases) both look rather clumsy, but are done that way so that the macro expands into an expression which is syntactically a simple command. The definition for the non-debugging case I took from `/usr/include/assert.h`; I do not recall where I got the definition for the debugging case, but the definition in `/usr/include/assert.h` looked to be gcc specific.

The purpose of the procedure `AssertionFailed` is explained above in the user documentation (to facilitate interception of failed assertions). The procedure never returns; instead it throws a CoCoALib exception with code `ERR::AssertFail`. Before throwing the exception it prints out a message on `std::cerr` summarising what the assertion was, and where it was. Note the non-standard way of throwing the CoCoA exception: this allows the `ErrorInfo` object to refer to the file and line where `CoCoA_ASSERT` was called (rather then to the line in `assert.C` where `CoCoA_ERROR` is called). The entire printed message is assembled into an `ostringstream` before being printed to ensure exception safety: either the whole message is printed or none of it is, since the printing step is an atomic operation.

7.4 Bugs, Shortcomings, and other ideas

Is the exception safe implementation of `AssertionFailed` excessive?

You have to use explicitly `#ifdef CoCoA_DEBUG` if you want to have a loop or any other non-trivial piece of code executed only when debugging it turned on.

The following (simplified but real) code excerpt is mildly problematic:

```
{
  bool OK = ....;
  CoCoA_ASSERT(OK);
}
```

When compiled without debugging (*i.e.* `CoCoA_DEBUG` is zero) the compiler (gcc-3) complains that the variable `OK` is unusued. It does not appear to be possible to make the macro "depend on its argument" in the non-debugging case without incurring the run-time cost of evaluating the argument (if the argument is just a variable the cost is negligible, but if it is a more complex expression then the cost could be considerable). The solution adopted was to modify the calling code like this:

```
{
  bool OK;
  OK = ....;
  CoCoA_ASSERT(OK);
}
```

Note that the apparently simpler code below **will not work** if the elided code (*i.e.* the `....`) has a side effect since the elided code will not be called at all in the non-debugging case:

```
{
  CoCoA_ASSERT(....);
}
```

POSSIBLE SOLUTION: maybe `CoCoA_ASSERT` could compute `sizeof(...)` in the non-debugging case – this should avoid evaluation of the argument, and will compile away to nothing.

8 BigInt (John Abbott)

8.1 Examples

- ex-BigInt1.C
- ex-BigInt2.C

- ex-BigInt3.C
- ex-GMPAllocator1.C
- ex-GMPAllocator2.C

8.2 User documentation

8.2.1 Generalities

The class `BigInt` is intended to represent (signed) integers of practically unlimited range; it is currently based on the implementation in the GMP *big integer* library. This code forms the interface between CoCoALib and the big integer library upon which it relies. It seems most unlikely that GMP will be displaced from its position as the foremost library for big integer arithmetic; as a consequence the class `BigInt` may eventually be replaced by GMP's own C++ interface.

The usual arithmetic operations are available with standard C++ syntax but generally these incur run-time overhead since results are returned through temporaries which are created and destroyed silently by the compiler. Thus if the variables `a`, `b` and `c` are each of type `BigInt` then `a = b+c;` is a valid C++ statement for placing the sum of `b` and `c` in `a`, **but** the sum is first computed into a hidden temporary which is then copied to `a`, and then finally the temporary is destroyed.

There is an important exception to the natural syntax: `^` does **not** denote exponentiation; you must use the function `power` instead. We have chosen not to define `operator^` to perform exponentiation because it is too easy to write misleading code: for instance, `a*b^2` is interpreted by the compiler as `(a*b)^2`. There is no way to make the C++ compiler use the expected interpretation.

A single arithmetic operation and assignment may be effected slightly faster using a less natural notation; this approach avoids using the hidden temporaries required with the natural notation. Thus instead of `a = b+c;` one can write `add(a, b, c);`. The reason for offering both syntaxes is to allow simpler and more natural code to be written for first versions; the time-critical parts can then be recoded using the faster but less natural notation (after suitable profiling tests, of course).

Arithmetic may also be performed between a `BigInt` and a machine integer. The result is always of type `BigInt` (with the sole exception of remainder by a machine integer). Do remember, though, that operations between two machine integers are handled directly by C++, and problems of overflow can occur.

It is important not to confuse values of type `BigInt` with values of type `RingElem` (Sec.71) which happen to belong to the ring `RingZZ` (Sec.79). In summary, the operations available for `RingElem` (Sec.71) are those applicable to elements of any ordered commutative ring, whereas the range of operations on `BigInt` values is wider (since we have explicit knowledge of the type).

8.2.2 The Functions Available For Use

Constructors

A value of type `BigInt` may be created from:

- nothing, in which case the value is zero
- another value of type `BigInt`
- a machine integer
- a string containing the decimal digits
- a GMP `mpz_t` value; in this case the ctor call must have first argument `CopyFromMPZ` and the second argument is the `mpz_t` value to be copied

No constructor for creating a `BigInt` from a `std::string` (or a `char*`) is provided. This is for two reasons: (A) a technical ambiguity in `BigInt(0)` since `0` is valid as a `char*`; (B) conversion from a decimal string representation is sufficiently costly that it should be highly visible. Conversion from a string to a value of type `BigInt` can be effected using the `convert` function (see `convert`) or using `operator>>` and `std::istringstream` from the C++ library.

Operations

See `IntOperations` (Sec.41)

1. Functions violating encapsulation
   - `mpzref(n)` – this gives a (const) reference to the `mpz_t` value inside a `BigInt` object. You should use this accessor very sparingly (but it is handy for calling GMP functions directly).

8.3 Maintainer Documentation

The implementation is structurally very simple, just rather long and tedious. The value of a `BigInt` object is represented as an `mpz_t`; this is a private data member, but to facilitate interfacing with code which uses `mpz_t` values directly I have supplied the two functions called `mpzref` which allow access to this data member.

The output function turned out to be trickier than one might guess. Part of the problem was wanting to respect the `ostream` settings.

Of course, input is a mess. Nothing clever here.

Check also the documentation for `MachineInt` (Sec.46) to understand how that class is used.

8.4 Bugs, shortcomings and other ideas

Currently functions which return `BigInt` values will copy the result (upon each return) – an attempt to avoid the waste with proxy classes caused a problem see

The official GMP interface is certainly more efficient, so the CoCoA library will presumably eventually switch to using GMP directly.

No bit operations: bit setting and checking, and/or/xor/not.

The code is long, tedious and unilluminating. Are there any volunteers to improve it?

8.5 Main changes

2012

- May (v0.9951):
  - moved common operations on `BigInt` (Sec.8) and `MachineInt` (Sec.46) together into `IntOperations` - **2011**
- August (v0.9950):
  - class `ZZ` renamed into `BigInt`: avoid confusion with `RingZZ` (Sec.79) and its name in CoCoA system
  - `random` has changed (was `random(lo,hi)`): see `RandomZZStream` (Sec.**??**), `RandomLongStream` (Sec.**??**)

9 BigRat (John Abbott)

9.1 Examples

- ex-BigRat1.C

9.2 User documentation

9.2.1 Generalities

The class `BigRat` is intended to represent (exact) rational numbers of practically unlimited range; it is currently based on the implementation in the GMP *big integer* library. This code forms the interface between CoCoALib and the big integer/rational library upon which it relies. It seems most unlikely that GMP will be displaced from its position as the foremost library of this type; as a consequence the class `BigRat` may eventually be replaced by GMP's own C++ interface.

The usual arithmetic operations are available with standard C++ syntax but generally these incur run-time overhead since results are returned through temporaries which are created and destroyed silently by the compiler. Thus if the variables `a`, `b` and `c` are each of type `BigRat` then `a = b+c;` is a valid C++ statement for placing the sum of `b` and `c` in `a`, **but** the sum is first computed into a hidden temporary which is then copied to `a`, and then finally the temporary is destroyed. As a general principle, the type `BigRat` is provided for convenience of representing rational values rather than for rapid computation.

There is an important exception to the natural syntax: `^` does **not** denote exponentiation; you must use the function `power` instead. We have chosen not to define `operator^` to perform exponentiation because it is too easy to write misleading code: for instance, `a*b^2` is interpreted by the compiler as `(a*b)^2`. There is no way to make the C++ compiler use the expected interpretation.

Arithmetic may also be performed between a `BigRat` and a machine integer or a `BigInt` (Sec.8). The result is always of type `BigRat` (even if the value turns out to be an integer). Do remember, though, that operations between two machine integers are handled directly by C++, and problems of overflow can occur.

It is important not to confuse values of type `BigRat` with values of type `RingElem` (Sec.71) which happen to belong to the ring `RingQQ` (Sec.76). The distinction is analogous to that between values of type `BigInt` (Sec.8) and value of type `RingElem` (Sec.71) which happen to belong to the ring `RingZZ` (Sec.79). In summary, the operations available for `RingElem` (Sec.71) are those applicable to elements of any ordered commutative ring, whereas the range of operations on `BigRat` values is wider (since we have explicit knowledge of the type).

9.2.2 The Functions Available For Use

Constructors

A value of type `BigRat` may be created from:

- nothing, in which case the value is zero
- another value of type `BigRat`
- `BigRat(n,d)` a pair of integers (machine integers or `BigInt` (Sec.8)s) specifying numerator and denominator in that order; you may supply a third argument of `BigRat::AlreadyReduced` if you are **absolutely certain** that there is no common factor between the given numerator and denominator
- a string of the form `N` or `N/D` where `N` is the decimal representation of the numerator and `D` that of the denominator
- `BigRat(mpq_value)` copy a GMP rational (of type `mpq_t`) into a `BigRat`; helps interfacing between CoCoALib and code using GMP directly.

See **Bugs** section for why there is no ctor from a single integer, and also for why `BigRat(0)` is accepted by the compiler.

Infix operators

1. normal arithmetic (potentially inefficient because of temporaries)
   - `+` the sum
   - `-` the difference
   - `*` the product
   - `/` floor quotient (divisor must be positive)
   - `=` assignment
2. arithmetic and assignment
   - `+=`, `-=`, `*=`, `/=`, `%=` – definitions as expected; LHS must be of type `BigRat`
3. arithmetic ordering
   - `==`, `!=`
   - `<`, `<=`, `>`, `>=` – comparison (using the normal arithmetic ordering) – see also the `cmp` function below.

4. increment/decrement
   - `++`, `--` (prefix, e.g. `++a`) use these if you can
   - `++`, `--` (postfix, e.g. `a++`) avoid these if you can, as they create temporaries

More functions

1. query functions (all take 1 argument)
   - `IsZero(q)` – true iff `q` is zero
   - `IsOne(q)` – true iff `q` is 1
   - `IsMinusOne(q)` – true iff `q` is -1
   - `IsOneNum(q)` – true iff `num(q)` is 1
   - `IsOneDen(q)` – true iff `den(q)` is 1
   - `sign(q)` – gives -1 (machine integer) to mean `q` is negative, 0 (machine integer) to mean `q` is zero, +1 (machine integer) to mean `q` is positive.
2. Exponentiation
   - `power(a, b)` – returns `a` to the power `b` (result is always a `BigRat`)
3. The cmp function (three way comparison)
   - `cmp(a, b)` – returns an `int` which is $<$ `0` if `a` $<$ `b`, or `==` `0` if `a == b`, or $>$ `0` if `a` $>$ `b`.
4. Other functions
   - `abs(q)` – gives the absolute value of `q`
   - `floor(q)` – returns a `BigInt` (Sec.8) for the greatest integer `<= q`
   - `ceil(q)` – returns a `BigInt` (Sec.8) for the least integer `>= q`
   - `round(q)` – returns a `BigInt` (Sec.8) which is the nearest to `q` (in case of ambiguity it rounds towards +infinity)
   - `num(q)` – returns a `BigInt` (Sec.8) which is the numerator of `q`
   - `den(q)` – returns a positive `BigInt` (Sec.8) which is the denominator of `q`
   - `log(q)` – returns a double whose value is (approx) the natural logarithm of `q`
5. Functions violating encapsulation
   - `mpqref(n)`– this gives a (const) reference to the `mpq_t` value inside a `BigRat` object. You should use this accessor very sparingly!

9.3 Maintainer Documentation

Nothing very clever. Conversion from a string was a bit tedious.

Note that the ctor call `BigRat(0)` actually calls the ctor from a string. Unfortunately, this a C++ "feature". It will result in a run-time error.

I have replaced the bodies of the `BigRat` ctors which take two integers as arguments by a call to the common body `BigRat::myAssign`. This does mean that some wasteful temporaries are created when either of the arguments is a machine integer. Time will tell whether this waste is intolerable.

9.4 Bugs, Shortcomings and other ideas

This code is probably not *exception safe*; I do not know what the `mpq_*` functions do when there is insufficient memory to proceed. Making the code *exception safe* could well be non-trivial: I suspect a sort of `auto_ptr` to an `mpq_t` value might be needed.

Removed `BigRat` ctors from a single (machine) integer because too often I made the mistake of writing something like `BigRat(1/2)` instead of `BigRat(1,2)`.

Should the `BigRat` ctor from string also accept numbers with decimal points? e.g. BigRat("3.14159")? We'll wait and see whether there is demand for this before implementing; note that GMP does **not** offer this capability.

Should the `BigRat` ctor from a string (or C string) also accept an optional `ReduceFlag`?

9.5 Main changes

2011

- August (v0.9950): class `QQ` renamed into `BigRat`: to avoid confusion with `RingQQ` (Sec.76) and its name in CoCoA system

10 bool3 (John Abbott)

10.1 User documentation for bool3

The class called `bool3` implements a three-valued boolean: the possible values represent the notions *false*, *uncertain* and *true*. A variable of type `bool3` has a default initial value of *uncertain*. To avoid problems with reserved words the three truth values are actually called:

`true3` `false3` `uncertain3`

10.1.1 Examples

- ex-bool3.C

10.1.2 Constructors

- `bool3(true)` – is the same as `true3`
- `bool3(false)` – is the same as `false3`

To convert a normal `bool` to a `bool3` value, you must call the ctor explicitly.

Nevertheless, a variable of type `bool3` may be assigned a C++ `bool` value (*e.g.* `bool3 b3 = true;`) in which case `true` maps to `true3` and `false` to `false3`.

10.1.3 Queries

There are three functions for testing the value of a `bool3` expression: (note that these functions return a C++ `bool` value)

- `IsTrue3(bool3expr)` – true iff expr is true3
- `IsFalse3(bool3expr)` – true iff expr is false3
- `IsUncertain3(bool3expr)` – true iff expr is uncertain3

These functions are the only way of *converting* a `bool3` to a standard C++ `bool` value – there is no automatic conversion from a `bool3` value to a standard C++ `bool`.

10.1.4 Operations on bool3

There are **no arithmetic operations** on `bool3` values.

`bool3` values may be printed in the usual way. The printed forms are: `true3` `false3` `uncertain3`.

10.1.5 Comparison with BOOST library

Note that `bool3` is not the same as BOOST's `tribool`, though the two are fairly similar. The principal differences are that `bool3` does not have automatic conversion to `bool`, and there are no logical operations on `bool3` whereas `tribool` does have some.

10.2 Maintainer documentation for bool3

The implementation is very simple. The only point to watch is that the order of the constants in the enum `Bool3TruthValues` was chosen to allow a simple implementation of the function `cmp` (which is currently removed from `bool3.H`, see *Bugs and Shortcomings* below). If you change the order, you will have to change the definition of `cmp`.

All functions/operations are implemented inline except for I/O. I have avoided const-ref arguments since it is surely cheaper simply to copy the enum value.

10.3 Bugs, Shortcomings and other ideas

I made the `bool3` ctor from `bool` explicit; if conversion from `bool` to `bool3` is automatic then machine integer value match `bool3` as well as they match `MachineInt` – be careful.

I do feel quite uneasy about disagreeing with BOOST's `tribool` design, but their example of a three-way *if* statement looks to me to be a recipe for programmer grief – one has to suppress the *law of the excluded middle* to read their code without finding it odd and surprising.

Boolean arithmetic operations are not defined since we have not needed them so far. It would be a simple matter, but I prefer to wait until there is a real need for such operations.

Is the `cmp` function ever going to be useful??? There was also a function `cmp` for comparing two `bool3` values:

```
 cmp(b1, b2)  returns an int <0, =0 or >0 according as b1 <,=,> b2
```

(assuming this ordering: false3 < uncertain3 < true3)

```
>     friend int cmp(bool3 lhs, bool3 rhs); // must be friend function
>   inline int cmp(bool3 lhs, bool3 rhs)
>   {
>     return lhs.myTruthValue - rhs.myTruthValue;
>   }
```

11 BuildInfo (John Abbott)

11.1 Examples

- ex-BuildInfo.C
- ex-limits.C

11.2 User documentation

The constant in `BuildInfo` allows you to find out which version of CoCoALib you are using. The function `BuildInfo::PrintAll` prints out all the build information on the `ostream` passed in as argument – you should include this information whenever you report a bug.

There is one string constant which contains the version number of the library: it is of the form `A.bcde` where `A` is the major version, `bc` is the minor version, and `de` is the patch level. Note that there are always precisely 4 digits after the point (even if they are all zero).

- `BuildInfo::version` – a C string containing the CoCoALib version number.

NOTE: if you happen upon a copy of libcocoa.a and want to find out which version it is, you can use the following Unix/Linux command:

```
strings libcocoa.a | egrep "CoCoA::BuildInfo"
```

This should print out three lines informing you of the library version, the compiler used, and the compiler flags used when creating libcocoa.a.

11.3 Maintainer documentation

I chose to put the constants and function in their own namespace to emphasise that they go together.

There are actually four string constants, but only one is supposed to be publicly accessible (because I cannot imagine why anyone would want access to the other three). I made the constants C strings because it seemed simpler than using C++ strings. The three constants `VersionMesg`, `CompilerMesg`, and `CompilerFlagsMesg` contain the substring `CoCoA::BuildInfo` so that the egrep trick described above will produce the version/compiler information directly.

I made `BuildInfo::PrintAll` leave a blank line before and after its message so that it would stand out better from other output produced by the program.

11.4 Bugs, Shortcomings and other ideas

The constants are not C++ strings – is this really a bug?

Should the three constants `VersionMesg`, `CompilerMesg`, and [CompilerFlagsMesg] be hidden or public? Until someone convinces me there is a good reason to make them public, they'll stay private.

12 CanonicalHom (John Abbott)

12.1 User Documentation for CanonicalHom

The function `CanonicalHom` can be used to create certain simple canonical homomorphisms. If it is unable to produce the required homomorphism then it will throw an exception of type `ErrorInfo` having error code `ERR::CanonicalHom` (see `error` (Sec.25)).

12.1.1 Examples

- ex-RingHom1.C
- ex-RingHom2.C
- ex-RingHom5.C

12.1.2 Constructors

In all cases the syntax is

- `CanonicalHom(domain, codomain)`

You can use `CanonicalHom` whenever the domain is `RingZZ` (Sec.79) or `RingQQ` (Sec.76), or if `codomain` is formed from `domain` in a single step. Here is a complete list of the cases when `CanonicalHom` will work:

- if `domain == codomain` then result is `IdentityHom`
- if `domain` is `RingZZ` (Sec.79) then result is `ZZEmbeddingHom`
- if `domain` is `RingQQ` (Sec.76) then result is `QQEmbeddingHom` (may be a **partial hom**)
- if `codomain == FractionField(domain)` then result is fraction field `EmbeddingHom`
- if `domain == CoeffRing(codomain)` then result is `CoeffEmbeddingHom`
- if `codomain` is a quotient of `domain` then result is `QuotientingHom`

12.2 Maintenance notes for CanonicalHom

Structurally simple and rather tedious. It is *important* that the cases of the domain being `RingZZ` (Sec.79) or `RingQQ` (Sec.76) are tested last because the other cases offer shortcuts (compared to `ZZEmbeddingHom` and `QQEmbeddingHom`).

12.3 Bugs, Shortcomings, etc

JAA does not like the structure of the code. Also the restriction to a "single step" seems artificial, but how to generalize this without perhaps producing annoying "semi-intelligent" code?

If you don't like `goto`, have a go at rewriting the implementation. I'll accept it so long as it is no more complicated than the current implementation!

Pity I cannot combine `IsPolyRing` and `AsPolyRing` to produce simpler code.

Are there any missing cases?

13 config (John Abbott)

13.1 User documentation for files config.H

The file config.H defines certain *global* concepts which may be used by any of the files in CoCoALib; in particular, this will include any definitions needed to ensure platform independence. Consequently, every header file in the CoCoA library should include the header file `CoCoA/config.H`.

The file `config.H` contains the following:

- typedefs for `SmallFpElem_t` and `SmallFpLogElem_t` which are used in `RingFpImpl` and `SmallFpImpl` (and their `Log` counterparts)
- typedef for `SmallExponent_t` which is used internally in some `PPMonoid` (Sec.58) implementations.

13.2 Maintainer documentation for files config.H and config.C

The typedef for `SmallFpElem_t` fixes the choice of representing type for elements in a `SmallFpImpl` which are used to implement a `RingFpImpl`; the type `SmallFpLogElem_t` does the same for `SmallFpLogImpl` and `RingFpLogImpl`. These types should be some size of unsigned integer; the best choices are probably platform dependent. If you want to try different choices, you will probably have to recompile the whole CoCoA library.

The typedef for `SmallExponent_t` should be an unsigned integer type. It is used in the `PPMonoid` (Sec.58)s which use an "order vector".

13.3 Bugs, Shortcomings, and other ideas

Putting `SmallFpElem_t` and `SmallFpLog_t` here is ugly. How can I do it better?

Shouldn't these typedefs be moved to the corresponding *.H files? What is the point of putting them here???

14 convert (John Abbott)

14.1 User Documentation for convert

The header file `convert.H` supplies several conversion functions and procedures. These functions/procedures are for converting a value of one type into another type (at least one of the types must be a CoCoALib type); the conversion may fail – failure is reported in different ways by different functions. There is also a way of safely converting machine integer values into other integral types.

There are two families of conversion functions:

1. `IsConvertible(dest,src)` the result is a boolean: `true` means the conversion was successful (and the result was placed in `dest`, the 1st arg)
2. `ConvertTo<DestType>(src)` the result is the converted value; if `src` cannot be converted then an error is thrown (with code `ERR::BadConvert`)

The `IsConvertible` functions attempt to convert the value to the type of the first arg; if the attempt succeeds, the converted value is assigned to the first arg and `true` is returned, otherwise `false` is returned (the value of the first arg is unchanged).

Here is a summary of the conversions currently offered:

"to" type	"from" type	notes
(unsigned) long	`BigInt` (Sec.8)	
(unsigned) int	`BigInt` (Sec.8)	
(unsigned) long	`BigRat` (Sec.9)	
(unsigned) int	`BigRat` (Sec.9)	
long	`RingElem` (Sec.71)	equiv to `IsInteger` & range check
`BigInt` (Sec.8)	`RingElem` (Sec.71)	same as `IsInteger`
`BigRat` (Sec.9)	`RingElem` (Sec.71)	same as `IsRational`
long	double	value must be integral & in range
`BigInt` (Sec.8)	double	
`BigRat` (Sec.9)	double	
double	`BigInt` (Sec.8)	may have rounding error!
double	`BigRat` (Sec.9)	may have rounding error!

Conversion to a `double` fails if overflow occurs. In contrast, underflow does not cause failure, and the converted value is simply 0.

There is a templated class called `NumericCast`; it is roughly analogous to `BOOST::numeric_cast`, and will eventually be replaced by direct use of this BOOST feature. It is to be used for converting safely from one machine integer type to another: the conversion succeeds only if the value supplied can be represented by the destination type. In case of failure an `ERR::BadConvert` exception is thrown.

14.2 Maintenance notes for convert

The `convert` procedures simply call the corresponding `IsConvertible` function – indeed a template implementation is appropriate here. A similar comment applies to the impl of `ConvertTo`; note that when calling a `ConvertTo` fn, it is necessary to specify only the first template type parameter (the compiler guesses the second parameter).

Only some combinations of `IsConvertible` functions are present so far.

The class `NumericCast` has a single template argument, and the constructor has a separate template argument to allow the "natural syntax" like that of `static_cast` (or BOOST's `numeric_cast`). I used a class rather than a templated function because a function would have required the user to specify two template arguments (*i.e.* unnatural syntax). I don't know if this is the best way to achieve what I want, but it is simple enough that there are *obviously no deficiencies*.

14.3 Bugs, Shortcomings, etc

The `IsConvertible` functions are a hotch potch, but how can it be done better?

BOOST has `numeric_cast` which is like `NumericCast` for built in numerical types. Sooner or later we should use that.

Should conversion to `double` ignore underflow, or should it say that the conversion failed?

15 debug-new (John Abbott)

15.1 User documentation

`debug_new.C` is distributed with CoCoALib, but is not really part of the library proper. Together with the standalone program `leak_checker` (Sec.44) it can help identify incorrect memory use (*e.g.* leaks). If you want to use `debug_new` to find a memory use problem, you may find it enough simply to see the section **Example** below.

The purpose of `debug_new` is to assist in tracking down memory use problems: most particularly leaks and writing just outside the block allocated; it is **not currently able** to help in detecting writes to deleted blocks. It works by intercepting all calls to global `new/delete`. Memory blocks are given small *margins* (invisible to the user) which are used to help detect writes just outside the legitimately allocated block.

`debug_new` works by printing out a log message for every memory allocation and deallocation. Error messages are printed whenever something awry has been found. The output can easily become enormous, so it is best to send the output to a file. All log messages start with the string

```
[debug_new]
```

and error messages start with the string

```
[debug_new] ERROR
```

so they can be found easily. Most messages include a brief summary of the amount of memory currently in use, the total amount allocated and deallocated, and the maximum amount of memory in use up to that point.

15.1.1 Finding memory leaks

To use `debug_new` to help track down memory leaks, you must employ the program called `leak_checker` (included in this distribution) to process the output produced by your program linked with `debug_new.o`. See `leak_checker` (Sec.44) for full details. Your program output should be put in a file, say called *memchk*. Then executing `leak_checker memchk` will print out a summary of how many `alloc/free` messages were found, and how many unpaired ones were found. The file *memchk* is modified if unpaired alloc/free messages were found: an exclamation mark is placed immediately after the word `ALLOC` (where previously there was a space), thus a search for `ALLOC!` will find all unpaired allocation messages.

Each call to `new/delete` is given a sequence number (printed as `seq=...`). This information can be used when debugging. Suppose, for instance, that `leak_checker` discovers that the 500th call to `new` never had a matching `delete`. At the start of your program (*e.g.* I suggest immediately after you created the `debug_new::PrintTrace` object) insert a call to

```
debug_new::InterceptNew(500);
```

Now use the debugger to set a breakpoint in `debug_new::intercepted` and start your program. The breakpoint will be reached during the 500th call to `new`. Examining the running program's stack should fairly quickly identify precisely who requested the memory that was never returned. Obviously it is necessary to compile your program as well as `debug_new.C` with the debugger option set before using the debugger!

Analogously there is a function `debug_new::InterceptDelete(N)` which calls `debug_new::intercepted` during the Nth call to `operator delete`.

15.1.2 Example

Try detecting the (obvious) memory problems in this program.

```
#include <iostream>
#include "CoCoA/debug_new.H"

int main()
{
  debug_new::PrintTrace TraceNewAndDelete; // merely activates logging of new/delete
  std::cout << "Starting main" << std::endl;
  int* pi1 = new int(1);
  int* pi2 = new int(2);
  pi1[4] = 17;
  pi1 = pi2;
  delete pi2;
  delete pi1;
  std::cout << "Ending main" << std::endl;
  return 0;
}
```

Make sure that `debug_new.o` exists (*i.e.* the `debug_new` program has been compiled). Compile this program, and link with `debug_new.o`. For instance, if the program is in the file `prog.C` then a command like this should suffice:

```
g++ -g -ICoCoALib/include prog.C -o prog debug_new.o
```

Now run `./prog >& memchk` and see the debugging messages printed out into *memchk*; note that the debugging messages are printed on `cerr/stderr` (hence the use of `>&` to redirect the output). In this case the output is relatively brief, but it can be huge, so it is best to send it to a file. Now look at the messages printed in *memchk*.

The *probable double delete* is easily detected: it happens in the second call to `delete` (`seq=2`). We locate the troublesome call to delete by adding a line in main immediately after the declaration of the `TraceNewAndDelete` local variable

```
debug_new::InterceptDelete(2); // intercept 2nd call to delete
```

Now recompile, and use the debugger to trap execution in the function `debug_new::intercepted`, then start the program running under the debugger. When the trap springs, we can walk up the call stack and quickly learn that `delete pi1;` is the culprit. We can also see that the value of `pi1` at the time it was deleted is equal the value originally assigned to `pi2`.

Let's pretend that it is not obvious why `delete pi1;` should cause trouble. So we must investigate further to find the cause. Here is what we can do. Comment out the troublesome delete (*i.e.* `delete pi1;`), and also the call to `InterceptDelete`. Recompile and run again, sending all the output into the file *memchk* (the previous contents are now old hat). Now run the `leak_checker` (Sec.44) program on the file *memchk* using this command: (make sure `leak_checker` has been compiled: `g++ leak_checker.C -o leak_checker`)

```
./leak_checker memchk
```

It will print out a short summary of the `new/delete` logs it has found, including a message that some unmatched calls exist. By following the instructions in `leak_checker` (Sec.44) we discover that the unfreed block is the one allocated in the line `... pi1 = new ...`. Combining this information with the *double delete* error for the line `delete pi1` we can conclude that the pointer `pi1` has been overwritten with the value of `pi2` somewhere. At this point `debug_new` and `leak_checker` can give no further help, and you must use other means to locate where the value gets overwritten (e.g. the *watch* facility of gdb; try it!).

WARNING `debug_new` handles **all** `new/delete` requests including those arising from the initialization of static variables within functions (and also those arising from within the system libraries). The `leak_checker` (Sec.44) program will mark these as unfreed blocks because they are freed only after main has exited (and so cannot be tracked by `debug_new`).

15.2 Maintainer documentation

This file redefines the C++ global operators `new` and `delete`. All requests to allocate or deallocate memory pass through these functions which perform some sanity checks, pass the actual request for memory management to `malloc/free`, and print out a message.

Each block requested is increased in size by placing *margins* both before and after the block of memory the user will be given. The size of these margins is determined by the compile-time (positive, even) integer constant `debug_new::MARGIN`; note that the number of bytes in a margin is this value multiplied by `sizeof(int)`. A margin is placed both before and after each allocated block handed to the user; the margins are invisible to the user. Indeed the user's block size is rounded up to the next multiple of `sizeof(int)` for convenience.

The block+margins obtained from the system is viewed as an integer array, and the sizes for the margins and user block are such that the boundaries are aligned with the boundaries between integers in the array – this simplifies the code a bit (could have used `chars`?). Each block immediately prior to being handed to the user is filled with certain values: currently `1234567890` is placed in each margin integer and `-999999999` is placed in each integer inside the user's block. Upon freeing, the code checks that the values in the margins are unchanged, thus probably detecting any accidental writes just outside the allocated block. Should any value be incorrect an error message is printed. The freed block is then overwritten with other values to help detect accidental "posthumous" read accesses to data that used to be in the block before it was freed.

For our use, the size of the block (the size in bytes as requested by the user) is stored in the very first integer in the array. A simplistic sanity check is made of the value found there when the block is freed. The aim is not to be immune to a hostile user, but merely to help track down common memory usage errors (with high probability, and at tolerable run-time cost). This method for storing the block size requires that the margins be at least as large as a machine integer (probably ought to use `size_t`).

Note the many checks for when to call `debug_new::intercepted`; maybe the code should be restructured to reduce the number of these checks and calls?

15.3 Shortcomings, bugs, etc

WARNING `debug_new` handles calls only to plain `new` and plain `delete`; it does not handle calls to `new(nothrow)` nor to `delete(nothrow)`, nor to any of the array versions.

Have to recompile to use the `debug_new::PrintTrace` to turn on printing. Maybe the first few messages could be buffered up and printed only when the buffer is full; this might buy enough time to bypass the set up phase of `cerr`?

Big trouble will probably occur if a user should overwrite the block size held in the margin of an allocated block. It seems extremely hard to protect against such corruption.

When a corrupted margin is found a printed memory map could be nice (compare with what `MemPool` (Sec.52) does).

An allocated block may be slightly enlarged so that its size is a whole multiple of `sizeof(int)`. If the block is enlarged then any write outside the requested block size but within the enlarged block is not detected. This could be fixed by using a `char` as the basic chunk of memory rather than an `int`. It is rather unclear why `int` was chosen, perhaps for reasons of speed? Or to avoid alignment problems?

Could there be problems on machines where pointers are larger than `int`s (esp. if the margin size is set to 1)? There could also be alignment problems if the margin size is not a multiple of the size of the type which has the most restrictive alignment criteria.

Is it right that the debugging output and error messages are printed on `cerr`? Can/Should we allow the user to choose? Using `cout` has given some trouble since it may call `new` internally for buffering: this seemed to yield an infinite loop, and anyway it is a nasty thought using the `cout` object to print while it was trying to increase an internal buffer.

The code does not enable one to detect easily writes to freed memory. This could be enabled by never freeing memory, and instead filling the freed blocks with known values and then monitoring for changes to these values in freed blocks. This could readily become very costly.

16 decimal (John Abbott)

16.1 User documentation for file decimal.H

These functions are to help visualize integer and rational numbers in a more comprehensible format. The `SigFig` argument is optional; the default value is 5.

- `MantissaAndExponent(N, SigFig)` splits the number `N` into three parts: the sign, the mantissa (with `SigFig` decimal digits), and the exponent If `N` is non-zero then the mantissa is a `BigInt` (Sec.8) whose value lies between `10^SigFig` and `10^(SigFig+1)-1` where both extremes are included. The result is a struct of type `MantExp` whose fields are `mySign`, `myMantissa` and `myExponent`.
- `FloatStr(N, SigFig)` convert the number `N` into a string of the form mantissa times power-of-ten, with `SigFig` digits in the mantissa. Note that trailing zeroes are not removed from the mantissa.
- `DecimalStr(N, SigFig)` convert the number `N` into a decimal string with `SigFig` digits. If `N` is very large or very small then it is passed to `FloatStr`. Note that trailing zeroes are not removed from the mantissa.

See also the functions `ILogBase` (in the doc for `BigInt` (Sec.8) and `BigRat` (Sec.9)).

16.2 Maintainer documentation

The functions `MantissaAndExponent` do the real work. The only tricky parts were deciding how to round in the case of a tie, and correct behaviour when the mantissa "overflows". I finally decided to round away from zero: it is easy to implement, and I wanted a solution which was symmetric about zero, so that `MantissaAndExponent` applied to `N` and to `-N` would always give the same result except for sign.

Mantissa overflow occurs only when the last digit rounds away from zero, and all the digits were 9 before rounding. This case would never occur if I'd chosen simply to truncate when forming the mantissa.

Printing of a `MantExp` structure is simple rather than elegant.

The function `DecimalStr` passes its args to `FloatStr` in two situations: if the number is so large that padding would be needed before the decimal point; if the number is so small that the `FloatStr` format would be shorter (i.e. if the exponent is less than -8).

16.3 Bugs, shortcomings and other ideas

The fields of a `MantExp` are publicly accessible; I'm undecided whether it is really better to supply the 3 obvious accessor fns.

The conversion in `MantissaAndExponent` is rather slow when the input number is large.

In principle the call to `ILogBase` could fail because of overflow; but in that case `ILogBase` itself should report the problem.

In principle a mantissa overflow could trigger an exponent overflow (*i.e.* if the exponent was already the largest possible long).

These functions cannot be applied directly to a machine integer; to call them you have to convert explicitly into a `BigInt` (Sec.8) (or `BigRat` (Sec.9)).

16.4 Main changes

2011

- February (v0.9943): first release

17 degree (John Abbott)

17.1 User documentation for the class degree

The class degree is used to represent the values returned by the "deg" function applied to power products and (multivariate) polynomials. Recall that in general a degree is a value in Z^k; the value of k and the way the degree is computed (equiv. weight matrix) are specified when creating the PPOrdering object used for making the PPMonoid of the polynomial ring – see the function NewPolyRing.

If t1 and t2 are two power products then the degree of their product is just the sum of their individual degrees; and naturally, if t1 divides t2 then the degree of the quotient is the difference of their degrees. The degree values are totally ordered using a lexicographic ordering. Note that a degree may have negative components.

The following functions are available for objects of type degree: degree d1(k); create a new degree object with value (0,0,...,0), with k zeroes

```
d1 = d2        assignment, d1 must be non-const
d1 + d2        sum
d1 - d2        difference (there might be no PP with such a degree)
d1 += d2       equivalent to d1 = d1 + d2
d1 -= d2       equivalent to d1 = d1 - d2
cmp(d1, d2)    (int) result is <0, =0, >0 according as d1 <,=,> d2
top(d1, d2)    coordinate-by-coordinate maximum (a sort of "lcm")
cout << d      print out the degree
GradingDim(d)  get the number of the components
d[s]           get the s-th component of the degree (as a BigInt) (for 0 <= s < k)
IsZero(d)      true iff d is the zero degree
SetComponent(d, k, n)  sets the k-th component of d to n (mach.int. or \texttt{BigInt} (Sec.\ref{sec:BigInt}
```

The six comparison operators may be used for comparing degrees (using the lexicographic ordering).

A `degree` object may be created by using one of the following functions:

```
  wdeg(f)     where f is a RingElem belonging to a \texttt{PolyRing} (Sec.\ref{sec:PolyRing})
  wdeg(t)     where t is a PPMonoidElem  (see \texttt{PPMonoid} (Sec.\ref{sec:PPMonoid}))
```

17.2 Maintainer documentation for the class degree

So far the implementation is very simple. The primary design choice was to use C++ std::vector<>s for holding the values – indeed a `degree` object is just a "wrapped up" vector of values of type `degree::ElementType`. For a first implementation this conveniently hides issues of memory management etc. Since I do not expect huge numbers of `degree` objects to created and destroyed, there seems little benefit in trying to use `MemPool` (Sec.52)s (except it

might be simpler to detect memory leaks...) I have preferred to make most functions friends rather than members, mostly because I prefer the syntax of normal function calls.

The `CheckCompatible` function is simple so I made it inline. Note the type of the third argument: it is deliberately not (a reference to) a `std::string` because I wanted to avoid calling a ctor for a `std::string` unless an error is definitely to be signalled. I made it a private static member function so that within it there is free access to `myCoords`, the data member of a `degree` object; also the call `degree::CheckCompatible` makes it clear that it is special to degrees.

As is generally done in CoCoALib the member function `mySetComponent` only uses CoCoA_ASSERT for the index range check. In contrast, the non-member fn `SetComponent` always performs a check on the index value. The member fn `operator[]` also always performs a check on the index value because it is the only way to get read access to the degree components. I used `MachineInt` (Sec.46) as the type for the index to avoid the nasty surprises C++ can spring with silent conversions between various integer types.

In implementations of functions on degrees I have preferred to place the lengths of the degree vectors in a const local variable: it seems cleaner than calling repeatedly `myCoords.size()`, and might even be fractionally faster.

`operator<<` **no longer** handles the case of one-dimensional degrees specially so that the value is not printed inside parentheses.

17.3 Bugs, Shortcomings and other ideas

The implementation uses `BigInt` (Sec.8)s internally to hold the values of the degree coordinates. The allows a smooth transition to examples with huge degrees but could cause some run-time performance degradation. If many complaints about lack of speed surface, I'll review the implementation.

Is public write-access to the components of a degree object desirable? Or is this a bug?

No special handling for the case of a grading over Z (i.e. k=1) other than for printing. Is this really a shortcoming?

Printing via `operator<<` is perhaps rather crude? Is the special printing for k=1 really such a clever idea?

`GradingDim(const degree&)` seems a bit redundant, but it is clearer than "dim" or "size"

Is use of `MachineInt` (Sec.46) for the index values such a clever idea?

18 DenseMatrix (John Abbott)

18.1 User documentation for dense matrices (and DenseMatImpl)

A normal user should never need to know about the class DenseMatImpl; see below for notes aimed at library maintainers.

An dense martrix object is a matrix represented in the most natural way: as a *2-dimensional array* of its entries. For instance a DenseMat of 4 rows and 3 columns will contain 12=4x3 entries. Contrast this with the a `SparseMatrix` where the values (and positions) of only the non-zero entries are recorded.

To create a `DenseMat` you need to specify its ring R and dimensions (`r` rows and `c` columns). By default the matrix is filled with zeroes; alternatively the entries may be initialized from a `vector` of `vector`.

```
NewDenseMat(R, r, c)    -- an r-by-c matrix filled with zero(R)
NewDenseMat(R, VV)      -- a matrix whose (i,j) entry is VV[i][j]
```

To create a copy of a matrix, MatrixView, ConstMatrixView use the call

```
NewDenseMat(M);
```

Currently a `DenseMat` has no special operations in addition to those for a general `matrix` (Sec.47). Here is a brief summary of those operations

```
BaseRing(M)        -- the ring to which the matrix entries belong
NumRows(M)         -- the number of rows in M (may be zero)
NumCols(M)         -- the number of columns in M (may be zero)

cout << M          -- print out the value of the matrix
```

```
M(i,j)             -- a copy of entry (i,j) in the matrix
SetEntry(M,i,j,value)  -- set entry (i,j) of matrix M to value
```

18.2 Maintainer documentation for the class DenseMatImpl

The implementation is really quite straightforward (apart from keeping proper track of `RingElemRawPtr`s when exceptions may occur).

`DenseMatImpl` is a concrete class derived from `MatrixBase` (see `matrix` (Sec.47)). As such it supplies definitions for all pure virtual functions. `DenseMatImpl` represents the value of a matrix as an object of type

```
vector< vector<RingElemRawPtr> >
```

The convention used is that the outer vector has an entry for each row, and each inner vector contains the values of that row. The indices of a matrix entry correspond directly to the `vector<>` indices needed to get at the value of that entry. The advantage of using a vector of vector is that resizing is relatively simple (compared to mapping the entries into a single vector whose length is equal to the total number of matrix entries).

Note that each entry in a `DenseMatImpl` is a `RingElemRawPtr`, so care must be taken to handle exceptions in a way which doesn't leak memory.

A `DenseMatImpl` object keeps explicit track of its own size (in the data members `myNumRows` and `myNumColumns`). This makes life easier when handling matrices one of whose dimensions is zero. The space overhead should normally be utterly negligible.

Member functions accepting indices use CoCoA_ASSERT to check the validity of the index values. This is useful during debugging but should cost nothing when compiled with debugging turned off.

18.3 Bugs and Shortcomings

Using `RingElemRawPtr` may not have been my brightest idea (because it becomes hard to make all functions fully exception clean).

The pseudo-ctor from vector of vector should probably be a template fn; this would offer better flexibility to the user (e.g. could initialize from a vector of vector of int).

This is a first implementation: simplicity was paramount, efficiency disregarded.

19 DenseUPolyClean (Anna Bigatti)

19.1 User documentation for files DenseUPoly.H and DenseUPoly.C

- `DenseUPolyRing NewPolyRing_DUP(const ring& CoeffRing)` – default indet name is x
- `DenseUPolyRing NewPolyRing_DUP(const ring& CoeffRing, const symbol& IndetName)`
- `DenseUPolyRing NewPolyRing_DUP(const ring& CoeffRing, const symbol& IndetName, long MinCapacity)`

19.2 Maintainer documentation for files DenseUPoly.H and DenseUPoly.C

The primary purpose for object of class `DenseUPoly` is to represent values in a `RingDenseUPoly`.

An object of type DenseUPoly is essentially a vector of coefficients (`RingElem`) and a degree (`long`).

In a valid `DenseUPoly` the vector has size at least min(1, deg). Moreover `coeff[deg]` is different from 0, the only exception being the zero polynomial, represented with `myCoeffsValue[0]=0` and `myDegValue=0`.

19.3 Bugs, Shortcomings, and other ideas

20 DenseUPolyRing (Anna Bigatti)

20.1 User documentation for DenseUPolyRing

`DenseUPolyRing` is an abstract class (inheriting from `PolyRing` (Sec.63)) representing rings of univariate polynomials written with *dense representation*: a polynomial is viewed abstractly as a vector of coefficients (belonging to the coefficient ring). Indices are `long`. All meaningful coefficients are in the positions 0 to `deg`. Between deg+1 and size-1 the coefficients are guaranteed to be valid and valued 0.

Zero polynomial is represented with `myDegPlus1 = 0`.

See `RingElem` DenseUPolyRing (Sec.71) for operations on its elements.

20.1.1 Pseudo-constructors

- `NewPolyRing_DUP(CoeffRing)` – default indet name is `x`
- `NewPolyRing_DUP(CoeffRing, IndetName)`
- `NewPolyRing_DUP(CoeffRing, IndetName, MinCapacity)` – allows to specify the minimum capacity for the coefficient vector (in order to minimize reallocations)

20.1.2 Query and cast

Let `R` be an object of type `ring` (Sec.69).

- `IsDenseUPolyRing(R)` – `true` if `R` is actually `DenseUPolyRing`
- `AsDenseUPolyRing(R)` – if `R` is a `DenseUPolyRing` *view* it as such

20.1.3 Operations on a DenseUPolyRing

See `PolyRing` operations (Sec.63).

20.2 Maintainer documentation for DenseUPolyRing

20.3 Bugs, Shortcomings and other ideas

21 DistrMPoly (John Abbott)

21.1 User documentation

21.2 Maintainer documentation

The primary purpose for object of class DistrMPoly is to represent values in a RingDistrMPoly; consequently all operations on a DistrMPoly which could allocate or deallocate memory must take arguments indicating which memory manager(s) to use.

An object of type DistrMPoly is essentially an ordered (singly) linked list of summands, and each summand is a pair made up of a coefficient (held as a RingElem) and a power product (held as a PPMonoidElem). In a valid DistrMPoly all the coefficients are non-zero, the power products are all distinct and the summands are ordered decreasingly according to the ordering on the PPs (as determined by the PPO inside the owning PPMonoid).

21.3 Bugs, Shortcomings, and other ideas

22 DistrMPolyInlPP (John Abbott)

22.0.1 User documentation for the class DistrMPolyInlPP

If you are not a CoCoA library developer then most likely you should not be considering using this class – use a polynomial ring instead. That way you will gain ease of use and safety with only a small performance penalty.

This class should be seen and used only by CoCoA library implementors; normal users should use the polynomial rings (which internally may well use DMPIs to represent their elements).

The class DistrMPolyInlPP implements multivariate polynomials represented as ordered linked lists of summands, each summand is made up of a coefficient (RingBase::RawValue) and a power product represented as an order vector.

The implementation is oriented towards speed rather than safety or ease of use. To this end argument sanity checks (and other types too) should normally use CoCoA_ASSERT rather than CoCoAError.

22.0.2 Maintainer documentation for the class DistrMPolyInlPP

NO DOCUMENTATION YET WRITTEN.

This code is very ugly, and I am far from happy with it. I hope it will eventually become much cleaner while maintaining the speed.

22.0.3 Bugs and Shortcomings

The biggest bug is the definition of the struct summand which is NOT PORTABLE since C++ compilers do not guarantee to respect the order of fields in a structure. I do not yet know how best to avoid this without incurring a run-time penalty.

This code is apparently stable but surprisingly ugly and awkward to use. I continue to hope that the same speed can be achieved with cleaner code. As an example of how bad the code is just take a look at the function deriv (which computes derivatives) – it is far too long and incomprehensible. There must be better way!

The classes DistrMPolyInlPP, PPMonoid and PPOrdering are more closely tied together than I would really like.

Iterators are giving me headaches – we need them, but they seem to expose implementation details.

23 DivMask (John Abbott)

23.1 Examples

- ex-DivMask1.C
- ex-DivMask2.C

23.2 User documentation

The main reason for creating a `DivMask` is to permit a quick, coarse test of divisibility between power products – but before you read on, you might like to consider using `PPWithMask` (Sec.62) instead, which offers essentially the same advantages with a *much more convenient interface*.

We say that `DivMask`s permit a "coarse" test because we accept as responses *definitely not divisible* or *possibly divisible* (but further checks must be conducted to decide for certain).

For example the radical of a PP .... (WORK-IN-PROGRESS)

`DivMask`s are a fairly low-level concept, and probably of little use to most normal CoCoALib users. If you need to do conduct a great many divisibility tests (between power products) and think you're interested, read on (assuming you have already decided that `PPWithMask` (Sec.62) does not fulfill your needs).

Note: currently `DivMask`s cannot be used to ascertain coprimality (see Bugs section).

To use DivMasks you must master two concepts. Firstly, the DivMask itself is simply a bitset wrapped up in a class. The size of the bitset is determined at compile time. There are various rules for how to set the bits in the bitset, but they all satisfy the following guiding principle:

> if t1 divides t2 then (DivMask(t1) & DivMask(t2)) == DivMask(t1)
>
> *i.e.* DivMask(t1) is a "subset" of DivMask(t2)

There are no other guarantees: in particular, the converse of the guiding principle does not hold in general.

23.2.1 Constructors and pseudo-constructors

You can create five different sorts of DivMaskRule:

WORK-IN-PROGRESS: explain what a DivMaskRule is

`NewDivMaskNull();` no bit is ever set (relatively fast, but otherwise pretty useless). (It is useful when a DivMaskRule is required and you know you won't use it)

`NewDivMaskSingleBit();` if the k-th exponent in the PP is strictly positive then the k-th bit is set: at most a single bit is used for each indeterminate, indets with `index >= DivMask::ourMaskWidth` are ignored completely.

`NewDivMaskSingleBitWrap();` if the k-th exponent in the PP is strictly positive then the `k%DivMask::ourMaskWidth`-th bit is set: all indets are taken into account, and each bit is used for a set of indeterminates. This implementation is good when we have many indeterminates in supposedly sparse PPs. (So far I don't have good examples with more than 2*ourMaskWidth indeterminates)

`NewDivMaskEvenPowers();` This rule may set several bits for a PP divisible by a "high" power of an indeterminate. For instance, with a mask width of 32 and 4 indets, up to 8 bits can be set for each indet: sets 1 bit if exponent is 1 or 2, set 2 bits if exponent is 3 or 4, etc. The actual number of bits set is `ceiling(exponent/2)`. This implementation is good when we have few indeterminates with high exponents (e.g. Buchberger's algorithm). It is equivalent to `SingleBitWrapImpl` if the number of indets is bigger than `ourMaskWidth`.

`NewDivMaskHashing();` this rule uses a hashing scheme to allow many bits to be set for each indet even when there are many indets. The number of bits set for an indet is `ceiling(sqrt(exponent))`.

Supposedly the implementation works well in all cases (e.g. few indets and high degrees, or many indets and low degrees)

For indet x the first bit set has index `x%ourMaskWidth`, and in general the k-th bit set has index `(x + k*hash2)%ourMaskWidth`. (See code for definition of hash2)

23.2.2 Operations

Operations with DivMaskRule

The type DivMaskRule is used to set the bits in a DivMask object. The possible function calls are:

- `DMR->myAssignFromExpv(mask, exps, NumIndets)` – sets mask according to PP with exponent vector exps. Currently the parameter exps must be of type `vector<SmallExponent_t>`, but this may change.

  This function might be quite expensive and its cost depends on the DivMaskRule, but this is not a problem if it is called much more rarely than IsSubset.

- `DMR->myOutputSelf(out)`

Operations with DivMask

The value of a DivMask object may be set any number of times (even using different DivMaskRules on each occasion). Any two DivMasks may be compared, but the result is meaningful only if both values were created using the same DivMaskRule.

There are a few comparison functions on DivMask objects – these are guaranteed to be very fast and independent of the DivMaskRule, unlike myAssignFromExpv

- `dm1 == dm2` – true iff the bitsets are equal
- `dm1 != dm2` – false iff the bitsets are equal
- `IsSubset(dm1, dm2)` – true if every bit set in dm1 is set in dm2

You can read the bits held inside a `DivMask` object using this function:

- `bits(dm)` – gives read-only access to the bitset inside the `DivMask`, the type of the result is `DivMask::mask_t` which is a typedef for a `std::bitset`.

23.3 Maintainer documentation

The class `DivMask` is pretty simple: we don't use a naked `bitset` to ensure that only a `DivMaskRule` can set the value. Use of bitwise-and for modular reduction restricts `ourMaskWidth` to being a power of 2. There are no member functions, and just one friend function (giving read access to the bitset):

```
friend const mask_t bits(const DivMask& dm);
```

The class `DivMaskRuleBase` is an abstract base class with an intrusive reference count: every concrete divmask rule must be derived from this class. The virtual member function `myAssignFromExpv` must be defined in each concrete divmask rule class: it should set the bits in the `DivMask` argument according to the exponents specified in the other two arguments. The virtual member function `myOutput` simply prints the name of the divmask rule – it might be useful during debugging. The protected member function `DivMaskRuleBase::myBits` simply allows write access to the `bitset` held inside a `DivMask` value; I have to do it this way because friendship is not inherited.

The type `DivMaskRule` is just a reference counting smart pointer to an instance of a concrete divmask rule class.

The entire declarations and definitions of the concrete classes are in the .C file. There is no need for them to be visible in the .H file.

The class `DivMaskNullImpl` is quite simple.

The class `DivMaskSingleBitImpl` is also very simple.

The class `DivMaskSingleBitWrapImpl` is implemented assuming that the mask width is a power of 2. It is quite simple.

The class `DivMaskEvenPowersImpl` was (half) written by Anna while under the influence of mind-altering drugs, I reckon.

The class `DivMaskHashingImpl` is a bit involved, especially regarding the choice of bits to set. I'm sure the heuristic can be improved (*e.g.* by actually trying it on some real cases :-) Currently the heuristic works as follows. We consider each indeterminate in turn: let `var` be the index of the indeterminate, and `exp` the exponent, then the total number of bits to be set is `ceil(sqrt(exp))`, and the first bit to be set will be in position `var%ourMaskWidth` and subsequent bits will be in positions separated by multiples of step (where step is `24*floor(var/ourMaskWidth)+13` – this was chosen because it happened to make `DivMaskHashingImpl` perform well in the CoCoALib tests).

23.4 Bugs, Shortcomings, and other ideas

Publicly visible use of `SmallExponent_t` is most unfortunate; how to fix it?

Define `operator<=` for DivMasks, to do the same as IsSubset??

Should default `ourMaskWidth` be 32 or 64? Surely most current processors are 64 bit now?

Is the restriction that `DivMask::ourMaskWidth` be a power of 2 reasonable? Would we really lose that much speed if any value were allowed? Chances are that the only interesting values are 32, 64 or 128 (which are indeed all powers of 2).

COPRIMALITY: Do we want `DivMask`s to permit a swift coprimality check? Presumably the idea would be that two disjoint DivMask values would imply that the corresponding PPs must be coprime. Another possibility is that the DivMask values are disjoint iff the PPs are coprime; this second possibility would exclude some ideas for implementing DivMasks (for instance `DivMaskSingleBitWrap` and `DivMaskHashing` would be excluded).

Documentation is too sarcastic.

23.5 Main changes

2006

- August: Removed almost all publicly visible references to SmallExponent_t; changed to long in all PPMonoid functions and SparsePolyRing functions. DivMask remains to sorted out.
- January: Added new DivMask type: DivMaskHashingImpl.

24 DynamicBitset (Anna Bigatti)

24.1 Examples

- ex-DynamicBitset1.C

24.2 User documentation

Class for representing square free monomials, or subsets of integers.

This is quite technical and useful only when efficiency is important.

Similar to a C++ `bitset` except that its size does not need to be fixed at compile time (hence the adjective *dynamic*).

24.2.1 Constructors

Let `n` be an integer, `pp` a `PPMonoidElem` (Sec.**??**), `b` a `DynamicBitset`

- `DynamicBitset(n)` – `DynamicBitset()` same as `DynamicBitset(0)`
- `DynamicBitset(ConstRefPPMonoidElem pp)`
- `DynamicBitset(const DynamicBitset&)`

24.2.2 Functions

Let `DB1` and `DB2` be two (const) values of type `DynamicBitset`

- `len(DB1)` – returns number of bits in `DB1`
- `count(DB1)` – returns number of set bits in `DB1`
- `out << DB1` – print out `DB1` (using currently chosen style)
- `DB1 | DB2` – bitwise or (equiv. the union of the subsets)
- `DB1 & DB2` – bitwise and (equiv. the intersection of the subsets)
- `DB1 - DB2` – bitwise diff (equiv. the set difference)
- `DB1 ^ DB2` – bitwise xor (equiv. union set-diff intersection)
- `IsSubset(DB1, DB2)` – true iff `DB1` is subset of `DB2`
- `IsDisjoint(DB1, DB2)` – true iff `DB1` and `DB2` are disjoint
- `Is1At(DB1, n)` – true iff `DB1` is 1 at position `n`
- `NewPP(PPM, DB1)` – create new PP in PPM whose exponents are given by `DB1`

24.2.3 Member functions

Additionally, let `DB` be a non-const value of type `DynamicBitset`.

- `DB1.myLen()` – number of bits
- `DB1.IamAll0s()` – true iff value is [00000...0000]
- `DB1.IamAll1s()` – true iff value is [11111...1111]

These two do not check that the index is valid:

- `DB.mySet(index, val)` – morally equiv to `DB[index] = val` (boolean)
- `DB.mySet(index)` – morally equiv to `DB[index] = true`
- `DB = DB1` – assignment
- `DB &= DB1` – equiv. to `DB = (DB & DB1)`
- `DB |= DB1` – equiv. to `DB = (DB | DB1)`
- `DB ^= DB1` – equiv. to `DB = (DB ^ DB1)`
- `DB -= DB1` – equiv. to `DB = (DB - DB1)`
- `DB1.Iam1At(index)` – equiv. to DB[index] == 1
- `bool operator<(const DynamicBitset& rhs) const;` – wrt Xel
- `DB1.IamSubset(DB2)` – true iff `DB1` is subset of `DB2`
- `DB1.IamDisjoint(DB2)` – true iff `DB1` and `DB2` are disjoint
- `DB1 == DB2` – true iff `DB1` and `DB2` have the same value
- `DB1 != DB2` – true iff `DB1` and `DB2` have different values

24.2.4 output options

Default printing style is `clean`, i.e. as an STL bitset of the same size. Printing style can be changed by setting the variable `DynamicBitset::ourOutputStyle` Example with a 66-bit `DynamicBitset` on a 64-bit machine:

`DynamicBitset::clean`	`000000000000000000000000000000000011`
`DynamicBitset::WithSeparators`	`00-00000000.00000000.00000000.00000011`
`DynamicBitset::AsRevVecOfLong`	`[0, 3]`

(see ex-DynamicBitset1.C).

Member functions

- `void myOutputSelf(std::ostream& out) const;` – as a bitset of same size
- `void myOutputSelf8(std::ostream& out) const;` – blocks of 8/ourNumBitsInBlock, for readability
- `void myOutputSelfLong(std::ostream& out) const;` – as reversed vector<unsigned long>

24.3 Maintainer documentation

Member fields (private)

`std::vector<BitBlock>`	`myVec;`
`unsigned long`	`mySizeValue;`

The `long` constant `DynamicBitset::ourNumBitsInBlock` stores number of bits contained in an `unsigned long` (normally 32 or 64).

So a `DynamicBitset` stores a STL vector of STL bitsets of (constant) size `ourNumBitsInBlock` called `myVec`. The field `mySizeValue` is the number of bits we intend to use. (*e.g.* in a 32 bit machine a `DynamicBitset` of size 60 is stored as a vector with 2 `BitBlock`s and will have 4 unused bits)

```
enum OutputStyle {clean, AsRevVecOfLong, WithSeparators};
```

Member functions (private)

- `myResize(long n);` – only for ctors
- `myVecLen() const;` – number of `BitBlock`s in vector

24.4 Bugs, shortcomings and other ideas

24.4.1 boost?

This class is needed because C++ `bitset` length has to be fixed at compile time. There is a class in boost named `dynamic_bitset`: if/when we decide CoCoALib inlude boost `DynamicBitset` will just call the boost implementation.

24.4.2 Stretchable?

`DynamicBitset`s, unlike boost's `dynamic_bitset`s, are not *stretchable*: the resize function is private. They are used to represent square-free power-products, therefore changing size does not make sense. But there is no technical reason to forbid it, so we might make it available.

24.5 Main changes

2010

- moved definition of class `facet` from `TmpIsTree` into `DynamicBitset.H,C` (and renamed). Rearranged and changed names for similarity with bitsets in STL and boost. Stuctured in *safe* or *fast* functions according to coding conventions. Test and example.

25 error (John Abbott)

25.1 Examples

- ex-error1.C

25.2 User documentation

25.2.1 Debugging

If you get a `CoCoA ERROR` when you execute your program, you can easily intercept it with your preferred debugger tool.

For example, when debugging with **gdb**, type

```
break CoCoA::ThrowError
```

and then `run`. When it stops in the call of `ThrowError`, type `up` to reach the line which originally caused the error.

25.2.2 Recommended way of reporting errors

Usually if you have detected an error in your program, you want to report it immediately. We recommend that you use the macro `CoCoA_ERROR` to do this. Here's an example:

```
value_t operator/(const value_t& num, const value_t& den)
{
  if (IsZero(den))
    CoCoA_ERROR(ERR::DivByZero, "operator/ for value_t");
  ....
}
```

The first argument should be an error ID (*i.e.* ERR::something); you can find a list of the IDs in the file (CoCoA_ROOT)/include/CoCoA/error.H. If no ID is suitable, you can just put a string instead. The second argument should be an indication of the function in which the error occurred.

25.2.3 Adding a New Error ID and its Default Message

If you are a CoCoALib contributor and want to add a new error ID and message (or even a new language for error messages), please read the maintainer documentation for what to do.

25.2.4 Information about errors – for the more advanced

The macro `CoCoA_ERROR` does two things:

- (1) it creates a `CoCoA::ErrorInfo` object with the parameters given to the macro, and also with the filename and line number;
- (2) it calls the function `CoCoA::ThrowError` on the `ErrorInfo` object just created.

Below we explain these two stages in more detail.

The class `CoCoA::ErrorInfo` is intended to be used for creating exception objects – indeed, it derives from `std::exception`. There are two things you are likely to want to do with exception objects:

- (A) create and throw the exception object
- (B) catch the exception object

- Case (A) Rather than using using the C++ `throw` command directly, we recommend that you pass the error object to the function `CoCoA::ThrowError` as it makes debugging easier (see above).

  We also recommend that you use the constructor which takes a `CoCoA::ERR::ID` and a string; the string should indicate where the error was detected, *e.g.* the name of the C++ function which found the error. Look through the list of `CoCoA::ERR::ID`s (in the file `error.H`) to find the one best suited to the type of error you wish to signal.

  If no error `CoCoA::ERR::ID` is suitable then you can use the constructor which accepts two string arguments: the first should be a description of the error (*e.g.* "Incompatible hypermatrix dimensions"), and the second should indicate where the error was detected. If you are a CoCoALib contributor, see the notes below about how to add a new error ID and message.

NOTE: if you set the C++ preprocessor symbol `CoCoA_DEBUG` to a value greater than 1 then a log message is produced each time `CoCoA::ThrowError` is called; the log message is printed on `std::clog`.

- Case (B) After catching a `CoCoA::ErrorInfo` object in the variable `err` you can make the following queries:

```
err == ERR::ErrorID  -- returns true iff err is of type ERR::ErrorID
                 (replace ErrorID by the ID of the error you want!)
err.what()  -- returns a C string being the error message;
```

EXAMPLE (of handling a CoCoA Error):

```
try { ... }
catch (const CoCoA::ErrorInfo& err)
{
  if (err != ERR::DivByZero) throw; // rethrow unexpected error
  // code to handle the "expected" division by zero error
}
```

If you have caught a `CoCoA::ErrorInfo` object and want to announce it as an error then call the procedure `CoCoA::ANNOUNCE` with the ostream on which to make the announcement and the ErrorInfo object as second argument. This will print an eye-catching error announcement, and then return to the caller. Note that `CoCoA::ANNOUNCE` does not cause the program to exit/abort, it merely prints out an eye-catching announcement.

To facilitate debugging, an `ErrorInfo` object may be printed in the usual way; this produces a modest message, clearly different from an error announcement.

Recall that, as for any other "exception object", simply creating a `CoCoA::ErrorInfo` object does not cause the error condition to be signalled. To signal the error it must be thrown – we recommend passing the error object to the function `CoCoA::ThrowError` (see above).

25.2.5 Choosing the language for error messages

You may choose the language for CoCoALib error messages: the default is English. If an error message has not yet been translated into the chosen language then it is replaced by the default english message. Currently there are only two choices:

```
ErrorLanguage::english();

ErrorLanguage::italian();
```

EXAMPLE:

```
int main()
{
  CoCoA::ErrorLanguage::italian(); // vogliamo messaggi d'errore in italiano
  ....
}
```

The language for error messages may be changed any number of times: the last chosen language is the one used when creating an `ErrorInfo` object.

25.3 Maintainer documentation for files error.H and error.C

`CoCoA::ErrorInfo` is derived from `std::exception` for compatibility with the rest of C++. How this might be useful I cannot yet say, but it does not measurably complicate the code (though it does force the implementation of a member function called `what`).

The preferred constructors for `ErrorInfo` are those accepting an `ERR::ID` and a C string indicating context (with or without filename and line number information); the other constructors should be used only when no suitable `ERR::ID` exists. The `ERR::ID` object indicates the general nature of the error, and is used for selecting the error message to be printed.

Note that the conversion from an `ERR::ID` to a string is slightly convoluted: this is to allow the possibility of selecting at run-time a language other than English for the error messages.

I chose not to offer an `ErrorInfo` constructor which accepts natively `const char*` args because the potential extra copying of strings (to construct a `std::string`) is hardly likely to be important, and `std::string`s feel much cleaner.

The nature and context of the error are kept separate in an `ErrorInfo` object since it is possible that we may wish to propagate the nature of the error to top level in an interactive system where it would be unhelpful or confusing to refer to some C++ function irrelevant to the user.

The definition of the function `CoCoA::ThrowError` is quite straightforward. The function is deliberately not inline: efficiency is wholly unimportant whereas the ability to set a breakpoint in the function is (some debuggers may be unable to set a breakpoint in an inline function).

Each CoCoA error ID object is in reality a constant global variable containing two pointers to constant C strings called `myName` and `myDefaultMesg`: the latter contains the associated default error message (which must be in English), and the former contains the name of the error ID object. The identity of the error ID actually resides in the address of the specific string constant in the data member `myName` – this implies that copying the ID object does not change its identity. Since the different objects necessarily have different names, the string literals containing those names are surely distinct, and so we are guaranteed that these addresses are distinct. There are comparison operators (equal, not-equal, and less-than) for `ERR::ID`; less-than is needed for using C++ maps when implementing error messages in languages other than english. These comparison operators merely conduct the required comparison on the addresses of the strings in `myName`; this is quick and simple, and sufficient for our purposes – the actual values of strings pointed to are not taken into consideration!

25.3.1 To Add a New Error Code and Message

Invent a new name for the error code, and insert it in the list of names of "error variables" (in the file error.H). Make sure you insert it in the right place respecting alphabetical order – this way it is easy to check whether a name is already present in the list. Add a short comment indicating what sort of error that code is to be used for.

Next you must add a message corresponding to that code. In the file error.C you will find a long list of "error variable" initializations. Add an initialization for your new "error variable" – the syntax is quite obvious from the other initializations there (which use the macro `DEFINE_ERROR`). You may wish to add translations of your new error message into the other languages present in error.C.

25.3.2 To Add a New Language for Error Messages

You must write a function analogous to the function italian() which resides inside the namespace `CoCoA::ErrorLanguage`. The new function must have a name different from the other functions there: I suggest the english name of the language. Inside the function you will have to fill a `MsgTable_t` object with the translated messages associated to each possible error code. At the end you should check to make sure that you have a message for each possible code: it should suffice simply to count them. The code will still compile and run even if some translated messages are missing: if an error occurs for which the current error language has no translation then the default (english) message is printed.

EXAMPLE: Suppose we want to add german error messages. We choose to use the name "german" for the function which activates german error messages. Here is what we do:

(1) edit error.H; immediately after the line containing "void italian();" insert "void german();"

(2) edit error.C; make a copy of the function italian(){...} and change its name to "german" – make sure you stay inside namespace `ErrorLanguage`; translate all the error messages in the strings.

25.4 Bugs, Shortcomings, and other ideas

The throw specifications on the destructor and `what` member function are needed for compatibility with `std::exception` – I regard this as a nuisance. I wonder if `std::string::c_str` can throw?

What about parameter values? In some cases it would be handy to give the bad value which caused the error: *e.g.* "Bad characteristic: 33". A problem is that a parameter value could be very large. We could simply allow up to 100 (say) characters of parameter information in a suitable string.

Only very few error messages have been translated into italian so far.

Perhaps allow the user to specify which ostream to print the logging message in `ThrowError`?

26 ExternalLibs-frobby (Anna Bigatti, Bjarke Hammersholt Roune)

26.1 User documentation

Frobby is a software system and project for computations with monomial ideals. **Frobby** is free software and it is intended as a vehicle for research on monomial ideals, as well as a useful practical tool for investigating monomial ideals.

Available functions:

```
long dimension(const ideal& I);

ideal AlexanderDualFrobby(I, pp);
ideal AlexanderDualFrobby(I);
ideal MaximalStandardMonomialsFrobby(I);

void IrreducibleDecompositionFrobby(std::vector<ideal>& components, I);
void PrimaryDecompositionFrobby(std::vector<ideal>& components, I);
void AssociatedPrimesFrobby(std::vector<ideal>& primes, I);

RingElem MultigradedHilbertPoincareNumeratorFrobby(I);
RingElem TotalDegreeHilbertPoincareNumeratorFrobby(I);
RingElem TotalDegreeHilbertPoincareNumeratorFrobby(I, const RingElem& base);
```

26.1.1 Examples

- ex-frobby1.C

26.1.2 Download and compile Frobby

frobby website

CoCoALib requires **Frobby** release 0.9.0 or later.

Download and compile **Frobby** following the instructions from the website. Then configure and compile **CoCoALib** typing

```
cd CoCoALib-0.99
./configure --with-libfrobby=<your_path_to>/libfrobby.a
make
```

NOTE: JAA says that to compile Frobby (0.8.2) on my machine I had to execute the following:

```
export CFLAGS="-m64 -mtune=core2 -march=core2"  # taken from gmp.h
export LDFLAGS=$CFLAGS
make
make library
```

26.2 Maintainer documentation

26.3 Bugs, shortcomings and other ideas

Currently Frobby is not really intended to be used as a library, so linking it with CoCoALib is not as simple as it could be. Hopefully this will soon change.

26.4 Main changes

2011

- 29 July: added (temporarily?) `Frobby` suffix to all functions
- 5 July: modified AlexanderDualFrobby into AlexanderDualFrobby, PrimaryDecomposition into PrimaryDecompositionFrobby.

2010

- 1 October: first inclusion

27 ExternalLibs-Normaliz (Anna Bigatti, Christof Soeger)

27.1 User documentation

Normaliz is a tool for computations in affine monoids, vector configurations, lattice polytopes, and rational cones.

Here we should include the manual for the normaliz flags/functions, but we wait until **libnormaliz** interface is more stable. For the moment look at the examples for available functions on `NormalizCone`s and setting flags.

27.1.1 Examples

- ex-Normaliz1.C
- ex-Normaliz2.C

27.1.2 Download and compile Normaliz

libnormaliz website

- **CoCoALib** (still at 29th July 2011) requires a more recent version than the official **Normaliz** release. The authors will be quite happy to send you a snapshot in the meanwhile.

  Download and compile **Normaliz** following the instructions from the website. Then configure and compile **CoCoALib** typing

  ```
  cd CoCoALib-0.99
  ./configure --with-libnormaliz=<your_path_to>/libnormaliz.a
  make
  ```

27.2 Maintainer documentation

27.3 Bugs, shortcomings and other ideas

27.4 Main changes

2011

- 26 July: new libnormaliz configuration (still a private copy)

2010

- 1 October: first inclusion

28 factor (John Abbott, Anna M. Bigatti)

28.1 Examples

- ex-factor.C

28.2 User documentation

There are several functions for computing factorizations of ring elements. The factorizations produced have different properties.

- `factor(f)` factorization into irreducibles
- `SqfreeFactor(f)` factorization into coprime squarefree factors
- `ContentfreeFactor(f)` polynomial factorization into (coprime) content-free factors

The irreducible factorization of a polynomial with rational coefficients produces factors with integer coefficients (and integer content = 1) having positive leading coefficient. The remaining factor is the unique rational number (actually a polynomial of degree 0) which makes the factorization correct.

28.3 Maintainer documentation

Still only a prototype – just uses old C4 code to do the work.

28.4 Bugs, shortcomings and other ideas

Still only a prototype – just uses old C4 code to do the work.

28.5 Main changes

2013

- aprile (v0.9953): first doc

29 factorization (John Abbott)

29.1 Examples

- ex-factor1.C

29.2 User documentation

In CoCoALib `factorization` is a template class with all fields public. Its purpose is to represent a (partial) factorization. The only operations for a `factorization` are the constructor and `operator<<` for printing.

A `factorization` contains three fields (currently they are all publicly accessible)

- `myFactors` contains a `std::vector` of factors found (may be empty)
- `myMultiplicities` contains a `std::vector` of the corresponding multiplicities (of type `long`), one for each factor found
- `myRemainingFactor` contains a remaining factor (which may be just a unit)

In CoCoALib there are just two instantiations of this template:

- `factorization<BigInt>` for the fns `factor` and `SmoothFactor` in `NumTheory`
- `factorization<RingElem>` for the fns `factor` and `SqfreeFactor` and `ContentFreeFactor` in `PolyRing` (actually `TmpFactor`)

The exact characteristics of the factors found depend on the function which generated the `factorization`. However the vectors in `myFactors` and `myMultiplicities` will be of the same length; the factors will be non-zero and non-invertible, and the multiplicities will be strictly positive.

29.2.1 Constructor

- `factorization(facs, mults, remfactor)` specifies initial values for the 3 fields

  The 3 fields are currently public, so that a factorization can be modified easily by the user (currently it is the user's responsibility to respect the conditions on the values).

29.3 Maintainer documentation

Short and simple! It's all in the header file.

29.4 Bugs, shortcomings and other ideas

It would be safer to have pairs of factor-and-multiplicity rather than two separate vectors whose length must be the same. However it may be less convenient for the user.

Add member functions for adjoining or removing factor-multiplicity pairs?

29.5 Main changes

2012

- October: chose "myMultiplicities" rather than "myExponents" as the field name.
- April: first version of doc (v0.9950)

30 FGModule (John Abbott)

30.1 User documentation for FGModule

`FGModule` is a reference counting smart pointer to an object of type `FGModuleBase`. Its value represents a Finitely Generated Module. Most modules in CoCoALib will probably actually be `FGModules`.

Let `v` be a `ModuleElem` belonging to an `FGModule`. Then we can access the various components of `v` using a syntax like that for indexing into a `std::vector`. Thus `v[n]` gives the `n`-th component (which will be a `RingElem`.

30.2 Examples

- ex-module1.C

30.3 Maintainer documentation for FGModule

Um.

30.4 Bugs, Shortcomings and other ideas

Documentation does not exist.

There was a suggestion to merge module.* with FGModule.* based on the reasoning that in practice all modules will (probably) be FGModules, so the distinction is rather pointless.

FGModule.C is jolly small – probably some code is missing.

31 FieldIdeal (John Abbott)

31.1 User documentation for files FieldIdeal*

The structure of ideals in a field is so simple that it is usually ignored completely: there are just two ideals being the zero ideal and the whole field. Nonetheless it is helpful to have an implementation of them.

There is only one publicly callable function here

```
NewFieldIdeal(k, gens)
```

where k is a CoCoA::ring which represents a field, and gens is a std::vector<CoCoA::RingElem> being a collection of generators (RingElem values belonging to k). It creates a CoCoA::ideal which represents the ideal of k generated by the elements of gens; gens may be empty.

For operations on ideals, please see `ideal` (Sec.39).

31.2 Maintainer documentation for files FieldIdeal*

The implementation is slightly more complex than one might naively expect: the primary complication is simply the need to retain the list of generators as specified by the user.

myTidyGensValue is either empty or it contains a single copy of RingElem(k, 1) according as the ideal is zero or the whole field.

31.3 Bugs, Shortcomings, and other ideas

One would like to think that code so short and so simple couldn't possibly harbour any nasty surprises. Then again one might just be surprised...

I definitely do not like the name of the function GetPtr; perhaps import? What is the correct way to achieve the end I want to achieve?

32 FractionField (John Abbott, Anna M. Bigatti)

32.1 User documentation for FractionField

A `FractionField` is an abstract class (inheriting from `ring` (Sec.69)) representing a fraction field of an effective GCD domain.

See `RingElem` FractionField (Sec.71) for operations on its elements.

32.1.1 Examples

- ex-RingQQ1.C
- ex-PolyRing1.C
- ex-RingHom5.C

32.1.2 Pseudo-constructors

- `NewFractionField(R)` – creates a new `ring` (Sec.69), more precisely a `FractionField`, whose elements are formal fractions of elements of `R` (where `R` is a true GCD domain, see `IsTrueGCDDomain` in `ring` (Sec.69)).
- `RingQQ()` – produces the CoCoA `ring` (Sec.69) which represents QQ, the field of rational numbers, fraction field of `RingZZ` (Sec.79). Calling `RingQQ()` several times will always produce the same unique ring in CoCoALib.

32.1.3 Query and cast

Let `S` be a `ring` (Sec.69)

- `IsFractionField(S)` – `true` iff `S` is actually a `FractionField`
- `AsFractionField(S)` – if `S` is a `FractionField` *view* it as such

```
if (IsFractionField(S))
{
   FractionField FrF = AsFractionField(S);
   ... code using FrF ...
}
```

  Calling `AsFractionField(S)` when `IsFractionField(S)` is `false` will throw an exception of type `CoCoAError` with code `ERR::NotFracField`

32.1.4 Operations on FractionField

In addition to the standard `ring` operations (Sec.69), a `FractionField` may be used in other functions.

Let `FrF` be a `FractionField` built as `NewFractionField(R)` with `R` a `ring` (Sec.69)

- `BaseRing(FrF)` – the `ring` (Sec.69) it is the `FractionField` of – an identical copy of `R`, not merely an isomorphic `ring` (Sec.69).

32.1.5 Homomorphisms

- `EmbeddingHom(FrF)` – `BaseRing(FrF)` –> `FrF`
- `InducedHom(FrF, phi)` – phi: `BaseRing(K)` –> `codomain(phi)`

32.2 Maintainer documentation for FractionField, FractionFieldBase, FractionFieldImpl

The class `FractionField` is wholly analogous to the class `ring` (Sec.69), *i.e.* a reference counting smart pointer. The only difference is that `FractionField` knows that the `myRingPtr` data member actually points to an instance of a class derived from `FractionFieldBase` (and so can safely apply a `static_cast` when the pointer value is accessed).

`FractionFieldBase` is an abstract class derived from `RingBase`. It adds a few pure virtual functions to those contained in `RingBase`:

```
virtual const ring& myBaseRing() const;
virtual ConstRawPtr myRawNum(ConstRawPtr rawq) const; // NB result belongs to BaseRing!!
virtual ConstRawPtr myRawDen(ConstRawPtr rawq) const; // NB result belongs to BaseRing!!
virtual const RingHom& myEmbeddingHom() const;
virtual RingHom myInducedHomCtor(const RingHom& phi) const;
```

`myBaseRing` returns a reference to the `ring` (Sec.69) (guaranteed to be an effective GCD domain) supplied to the constructor.

`myRawNum` (resp. `myRawDen`) produces a raw pointer to a value belonging to `BaseRing` (and *NOT* to the `FractionField`!); these two functions *practically* *oblige* the implementation of `FractionField` to represent a value as a pair of raw values "belonging" to the `BaseRing`. Note that, while the value belongs to `BaseRing`, the resources are owned by the `FractionField`!!

`EmbeddingHom` returns the embedding homomorphism from the `BaseRing` into the `FractionField`; it actually returns a reference to a fixed homomorpism held internally.

`InducedHom` creates a new homomorpism from the `FractionField` to another `ring` (Sec.69) S given a homomorpism from the `BaseRing` to S.

`FractionFieldImpl` implements a general fraction field. Its elements are just pairs of `RawValue`s belonging to the `BaseRing` (see the struct `FractionFieldElem`). For this implementation the emphasis was clearly on simplicity over speed (at least partly because we do not expect `FractionFieldImpl` to be used much). For polynomials whose coefficients lie in a `FractionField` we plan to implement a specific `ring` (Sec.69) which uses a common denominator representation for the whole polynomial. If you want to make this code faster, see some of the comments in the bugs section.

Important: while fractions are guaranteed to be reduced (*i.e.* no common factor exists between numerator and denominator), it is rather hard to ensure that they are *canonical* since in general we can multiply numerator and denominator by any unit. See a **bug comment** about normalizing units.

32.3 Bugs, Shortcomings and other ideas

The functions `myNew` are not *exception safe*: memory would be leaked if space for the numerator were successfully allocated while allocation for the denominator failed – nobody would clean up the resources consumed by the numerator. Probably need a sort of `auto_ptr` for holding temporary bits of a value.

Should partial homomorphisms be allowed: *e.g.* from QQ to ZZ/(3)? Mathematically it is a bit dodgy, but in practice all works out fine provided you don't divide by zero. I think it would be too useful (*e.g.* for chinese remaindering methods) to be banned. Given phi:ZZ->ZZ[x] it might be risky to induce QQ->ZZ[x]; note that ker(phi)=0, so this is not a sufficient criterion!

In fact one could create a `FractionFieldImpl` of any integral domain (it just wouldn't be possible to cancel factors without a GCD). I'll wait until someone really needs it before allowing it.

It is not clear how to make the denominator positive when the GCD domain is ZZ (so the fraction field is QQ). In general we would need the GCD domain to supply a *normalizing unit*: such a function could return 1 unless we have some special desire to normalize the denominator in a particular way. HERE'S A CONUNDRUM: FractionField(Q[x]) – all rationals are units, and so we could end up with horrible representations like (22/7)/(22/7) instead of just 1. MUST FIX THIS!!

The printing function is TERRIBLE!

FASTER + and - Addition and subtraction can be done better: let h be the GCD of the two denominators, suppose we want to compute a/bh + c/dh (where gcd(a,bh) = gcd(c, dh) = gcd(b,d) = 1 *i.e.* h = gcd(B,D) where B,D are the denoms) If h = 1 then there is no cancellation, o/w gcd(ad+bc, bdh) = gcd(ad+bc, h), so we can use a simpler gcd computation to find the common factor.

FASTER * and / Multiplication and division can also be made a little faster by simplifying the GCD computation a bit. The two cases are essentially the same, so I shall consider just multiplication. Assuming inputs are already reduced (*i.e.* there is no common factor between numerator and denominator). To compute (a/b)*(c/d), first calculate h1 = gcd(a, d) and h2 = gcd(b, c). The result is then: num = (a/h1)*(c/h2) & den = (b/h2)*(d/h1) and this is guaranteed to be in reduced form.

`myIsRational` is incomplete: it will fail to recognise rationals whose numerator and denominator have been multiplied by non-trivial units. BAD BUG! Ironically `myIsInteger` does work correctly.

33 FreeModule (John Abbott)

33.1 Examples

- ex-module1.C

33.2 User documentation for the class FreeModule

For normal use there are only three functions of interest:

NewFreeModule(R, NumCompts) creates an object of type FGModule representing the free module of dimension NumCompts over the ring R.

FreeModule(M) where M is a module; if M is genuinely a FreeModule then that FreeModule is returned otherwise an error is generated.

IsFreeModule(M) true iff the module M is genuinely a FreeModule.

NewFreeModule(R, NumCompts, shifts) creates an object of type FGModule representing the free module of dimension NumCompts over the ring R. R must be a PolyRing, and shifts is a vector<degree> containing NumCompts elements, the i-th element being the shift applied to degrees of values in the i-th component. For example: ?????

33.3 Maintainer documentation for the classes FreeModule and FreeModuleImpl

I shall suppose that the maintainer documentation for modules and FGModules has already been read and digested. It could also be helpful to have read ring.txt since the "design philosophy" here imitates that used for rings.

As one would expect, FreeModule is simple a reference counting smart pointer class to a FreeModuleImpl object.

FreeModuleImpl turns out to be a little more complex than one might naively guess. The extra complexity arises from two causes: one is compatibility with the general structure of modules, and the other is that a FreeModule manages the memory used to represent the values of ModuleElems belonging to itself.

GradedFreeModuleImpl is derived from FreeModuleImpl and allows storing and using ordering and shifts: it requires a SparsePolyRing as BaseRing. It provides these functions for FreeModule:

```
FreeModule NewFreeModule(const ring& P, const ModuleTermOrdering& O);
bool IsGradedFreeModule(const module& M);
```

The following functions are defined only if FreeModule is implemented as GradedFreeModuleImpl

```
const std::vector<degree>& shifts(const FreeModule& M);
const ModuleTermOrdering& ordering(const FreeModule& M);
long LPos(const ModuleElem& v);
```

```
degree wdeg(const ModuleElem& v);
int CmpWDeg(const ModuleElem& v1, const ModuleElem& v2);
ConstRefPPMonoidElem LPP(const ModuleElem& v);
bool IsHomog(const ModuleElem& v);
```

33.4 Bugs, Shortcomings and other ideas

Documentation rather incomplete.

34 GBEnv (Anna Bigatti)

34.1 User documentation

This class contains some information needed for the computation of a GBasis (with Buchberger's algorithm)

At the moment the file contains instead the class `GRingInfo` which was defined in `TmpGPoly.H`

One idea to unify the class of ideals in `SparsePolyRing` (Sec.87) is to make an abstract `GBMill` as a base for the the operation on ideals (toric, squarefree, ideals of points,..) For *standard* ideals the key for the (GB)operations is computing with Buchberger algorithm, therefore the `BuchbergerMill` should inherit from `GBMill`.

34.2 Maintainer documentation

As *one class should do one thing* `GRingInfo` and `GReductor` should reorganized and split into `GBEnv`, `GBInfo`, and `GBMill`.

Mill: *A building equipped with machinery for processing raw materials into finished products*

34.2.1 GBEnv will know

the environment for the arithmetics, that is:

- the `SparsePolyRing` involved
- the `DivMaskRule`
- the `PPMonoid` for `LPPForDiv`
- the `ring` of coefficients (field or FrFldOfGCDDomain)
- if it represents a module computation
- the "module/ring" embeddings (or GBHom ???)
- —> inheritance for the module case?

Notes

Embeddings/deembeddings are now in TmpGReductor: they embed polynomials and ModuleElems into `GPoly` (Sec.36)s therefore cannot directly be GBEnv member functions (i.e. GBEnv would need GPoly forward declaration or .H inclusion)

Should embeddings/deembeddings be a class on their own? or just functions in a file on their own? or where?

The main difference between ring and module computations is in considering the component in `IsDivisibleFast`. How to balance efficiency and inheritance? (The other difference is in making pairs of polynomials with the same component)

34.2.2 GBInfo will know

constant `GBEnv` and the flags related with the algorithm:

- if the input was homogeneous (for interreduction?)

- alg homog/aff/sat/hdriven...
- the kind of sugar function to use
- active criteria (Weyl/module). Now `GBCriteria` is in `GReductor`
- ...

34.2.3 GBMill/BuchbergerMill (?) will know – was GReductor

constant `GBInfo` and the "frozen computation":

- the input polynomials
- list of pairs
- list of reducers
- list for output
- reducing SPolynomial (or atomic ???)
- stats
- ...

Some general notes

Partial steps for the conversion of the current code:

1. use member function in ctor for common assignments (done)
2. transform functions with `GRingInfo` as argument into GRingInfo member functions (wip)

Good to know:

1. reduce.C uses only "env" info from `GRI.`
2. GRingInfo has many fields, completely unused in the basic case (ie GB for polys). Some are set with a default value which is different for some "concrete classes" (eg modules, operation specific)
3. SPoly creates a GPoly with "env" info, only sugar needs "mill"; in fact the constructor for GPoly needs "mill" only for sugar: we can make an "undefined" sugar to be set later on.

34.3 Bugs, shortcomings and other ideas

Everything so far is just work in progress.

34.4 Main changes

2010

- moved definition of class `GRingInfo` into `GBEnv.H,C`

35 geobucket (Anna Bigatti)

35.1 User documentation

A geobucket is a polynomial represented in a C++ vector of buckets: a `bucket` contains a polynomial (and some other info)

This construction is useful for **adding many short polynomials to a long one** (in particular the reduction process) because it lowers the number of calls of `cmp` between `PPMonoidElem`s.

35.1.1 Examples

- no examples yet

35.1.2 Constructors

- `geobucket(const SparsePolyRing&);`

35.1.3 Queries

- `IsZero(g)` – true iff `g` is the zero polynomial

35.1.4 Operations

Let `gbk` be a `geobucket`, `f` a `RingElem&` (see `RingElem` (Sec.71))

- `CoeffRing(gbk)` – the `ring` (Sec.69) of coefficients of the ring of `gbk`
- `PPM(gbk)` – the `PPMonoid` (Sec.58) of the ring of `gbk`
- `LC(gbk)` – the leading coeff of `gbk`; it is an element of `CoeffRing(gbk)`
- `content(gbk)` – the gcd of all coefficients in `gbk`; it is an element of `CoeffRing(gbk)`
- `RemoveBigContent(gbk)` – if `gbk` has a big content, `gbk` is divided by it
- `AddClear(f, gbk)` – assign the polynomial value of `gbk` to `f`, and set 0 to `gbk`
- `MoveLM(f, gbk);`
- `ReductionStep(gbk, f, RedLen);`
- `ReductionStepGCD(gbk, f, FScale, RedLen);`
- `operator<<(std::ostream&, gbk)`
- `PrintLengths(std::ostream&, gbk)` – just for debugging
- `operator<<(std::ostream&, const geobucket&)`

Member functions

- `myAddClear(f, len)`
- `myDeleteLM(void)`
- `myPushBackZeroBucket(MaxLen)`
- `myBucketIndex(len)` – the index for the `bucket` with length `len`
- `myAddMul(monom, g, std::gLen, SkipLMFlag)` – `*this += monom*g`
- `myDivByCoeff(coeff)` – content MUST be divisible by coeff
- `myMulByCoeff(coeff)`
- `myCascadeFrom(std::size_t i)`
- `mySize(void)`
- `mySetLM()` – Sets the LM of `*this` in the 0-th `bucket` and set `IhaveLM` to true; `*this` will be normalized

35.2 Maintainer documentation

After calling `gbk.mySetLM()` the leading monomial of `gbk` is in `gbk.myBuckets[0]` (and then `gbk` is zero iff `gbk.myBuckets[0]=0`)

`gbk.myBuckets[i]` contains at most `gbk_minlen * gbk_factor^i` summands

- `myPolyRing` – the SparsePolyRing gbk lives in
- `IhaveLM` – true if certified that LM(gbk) = LM(gbk[0])
- `myBuckets` – the bucket vector

35.2.1 bucket

This class is to be used only by `geobucket`s.

A `bucket` represents a polynomial as a product of a polynomial and a coefficient, two `RingElem` respectivey in a `SparsePolyRing` (Sec.87) `P` and `CoeffRing(P)`.

The coeffient factor is used for fast multiplication of a geobucket by a coefficient and it comes useful in the reduction process over a field of fraction of a GCD ring.

We normalize the `bucket` (i.e. multiply the polynomial by the coefficient) only when it is necessary: e.g. to compute a reference to the LC of the bucket.

All methods are private (to be used only by `geobucket`s, friend)

Methods on buckets (weak exception guarantee)

- `myNormalize(void)` – myPoly *=myCoeff; myCoeff 1
- `myAddClear(RingElem& f, int FLen)` – *this += f; f = 0; *this normalized
- `myAddClear(bucket& b)` – *this += b; b = 0; *this normalized
- `myMul(ConstRefRingElem coeff)` – *this *= coeff
- `myDiv(ConstRefRingElem coeff)` – *this /= coeff; assumes *this divisible by coeff

Functions on buckets

- `IsZero(const bucket&)` –
- `content(const bucket& b)` –
- `poly(bucket& b)` – normalize b and return a reference to the polynomial

Dirty method and function for efficiency (b1 and b2 will be normalized))

- `myIsZeroAddLCs(const SparsePolyRing&, bucket& b1, bucket& b2)` – `b1 += LM(b2); b2 -= LM(b2);` return `LC(b1)+LC(b2)==0`; it assumes `LPP(b1) == LPP(b2)`
- `MoveLM(const SparsePolyRing&, bucket& b1, bucket& b2)` – `b1 += LM(b2); b2 -= LM(b2);` it assumes `LPP(b1)<LPP(b2)`

Member fields

- `myPoly` – the polynomial (a `RingElem` (Sec.71) in `P`)
- `myCoeff` – the coefficient factor (a `RingElem` (Sec.71) in `CoeffRing(P)`)
- `myMaxLen` – the maximal length allowed for the polynomial of this bucket
- `myApproxLen` – an upper bound for the current length of the polynomial of this bucket

35.3 changes

2004

- October: introduction of `myDivMaskImplPtr` for computing `LPPwMask`: LPP with DivMask if this pointer is 0 LPPwMask returns an error (through `CoCoA_ASSERT`?)

36 GPoly (Anna Bigatti)

36.1 User documentation for the class GPoly

This part should be written by Massimo Caboara

36.2 Maintainer documentation for the class GPoly

Also this part should be written by Massimo Caboara, but, as I am the author and maintainer of the reduction code, I write some notes here.

A GPoly contains some member fields which often depend solely on the field myPolyValue: myLen, myWDeg, myLPPwMask, myComponent After a reduction we change the value of myPolyValue and the above fields can be updated accordingly calling: myUpdateLenLPPDegComp(); NB: if myPolyValue is 0 the fields myWDeg, myLPPwMask, myComponent are unreliable (intrinsicly undefined).

36.2.1 Old logs

GPoly.H

```
// Revision 1.14  2006/03/21 13:41:52  cocoa
// -- changed: removed typedef before enum CoeffEncoding::type
```

reduce.C

```
// Revision 1.20  2006/05/02 14:38:15  cocoa
// -- changed "and,or,not" to "&&,||,!" because of M$Windoze (by M.Abshoff)
//
// Revision 1.19  2006/04/27 13:35:57  cocoa
// -- reverted: using CmpLPP no faster that comparing LPP()
//
// Revision 1.18  2006/04/27 11:32:03  cocoa
// -- improved myReduceTail using CmpLPP
//
// Revision 1.17  2006/04/12 17:00:20  cocoa
// -- changed: myReduceTail does nothing if ( LPP(*this) < LPP(g) )
//     ==>  great speedup on 6x7-4_h
//
// Revision 1.12  2006/03/17 18:17:16  cocoa
// -- changed: use of ReductionCog for reduction (major cleanup)
//
// Revision 1.5  2004/03/04 11:38:28  cocoa
// -- updated code for Borel reductors:
//    "reduce" first checks for myBorelReductors and updates them when needed
```

37 GlobalManager (John Abbott)

37.1 Examples

- ex-empty.C – recommended structure for a program using CoCoALib

- ex-GMPAllocator1.C
- ex-GMPAllocator2.C

37.2 User Documentation

A `GlobalManager` object does some very simple management of certain global values used by CoCoALib. You **must create exactly one** object of type `GlobalManager` **before** using any other feature of CoCoALib. Conversely, the `GlobalManager` object must be destroyed only **after** you have completely finished using CoCoALib values and operations. An easy way to achieve this is to create a local variable of type `GlobalManager` at the start of a top level procedure (*e.g.* `main`) – see the CoCoALib example programs listed above.

Shortcut: most likely you will want to use one of the following at the start of your top-level procedure:

```
GlobalManager CoCoAFoundations;                       // use default settings
GlobalManager CoCoAFoundations(UseNonNegResidues); // printing preference
GlobalManager CoCoAFoundations(UseGMPAllocator);   // faster but NOT THREADSAFE!
```

Note about threadsafety the ctor for `GlobalManager` is not threadsafe; it is the user's responsibility to avoid trying to create several instances simultaneously.

37.2.1 Constructors and pseudo-constructors

The ctor for a `GlobalManager` has one (optional) argument. This argument is used to specify the global settings, namely

1. the type of memory manager to use for GMP values (*viz.* big integers and rationals), and
2. the convention for elements of rings of the form ZZ/m, *viz.* least non-negative residues or least magnitude (symmetric) residues.

The current defaults are to use the system memory mananger and symmetric residues.

Specifying the memory manager for BigInt values

CoCoALib `BigInt` (Sec.8) values are implemented using the GMP library which needs to have access to a memory manager. There are three possibilities for specifying the memory manager for GMP:

- `UseSystemAllocatorForGMP` (**default**) to use the system memory manager (*i.e.* `malloc`)
- `UseGMPAllocator` to use the CoCoALib custom memory manager
- `UseGMPAllocator(sz)` to use the CoCoALib custom memory manager with a slice size of *sz* bytes

WARNING if your program is multi-threaded or if you store GMP values in global variables or if your program uses another library which depends on GMP, then it is safest to use only the system memory manager!

Nevertheless, the CoCoALib custom allocator offers slightly better performance, and may be helpful when debugging or fine-tuning code.

Specifying the convention for modular integers

CoCoALib lets you choose between two conventions for printing elements of rings of the form ZZ/m:

- `UseSymmResidues` (**default**) symmetric residues (if m is even, the residue m/2 is printed as positive)
- `UseNonNegResidues` least non-negative residues (*i.e.* from 0 to m-1)

You may ask CoCoALib which convention has been chosen using `DefaultResiduesAreSymm()` see `GlobalManager` operations (Sec.37) below.

Combining several global settings

To specify more than one global setting the individual specifiers should be combined using `operator+`, like this:

```
GlobalManager CoCoAFoundations(UseNonNegResidues + UseGMPAllocator);
```

Combining incompatible or redundant specifiers will produce a run-time error: an exception of type `CoCoA::ErrorInfo` having `error` (Sec.25) code `ERR::BadGlobalSettings`.

Similarly an exception will be thrown if you attempt to create more than one live `GlobalManager` object. The exception is of type `CoCoA::ErrorInfo` and has `error` (Sec.25) code `ERR::GlobalManager2`.

37.2.2 Operations

Once the `GlobalManager` has been created you can use the following functions:

- `DefaultResiduesAreSymm()` – returns `true` iff the convention is `UseSymmResidues`.
- `GlobalRandomSource()` – returns a global randomness source; see `RandomSource` (Sec.66) for a description of the permitted operations on random source objects.

37.2.3 The Purpose of the GlobalManager

The concept of `GlobalManager` was created to handle in a clean and coherent manner (almost) all global values used by CoCoALib; in particular it was prompted by the decision to make the ring of integers a global value (and also the field of rationals). The tricky part was ensuring the orderly destruction of `RingZZ` (Sec.79) and `RingQQ` (Sec.76) before `main` exits. Recall that C++ normally destroys globals after `main` has completed, and that the order of destruction of globals cannot easily be governed; destroying values in the wrong order can cause to the program to crash just before it terminates. Another advantage of forcing destruction before `main` exits is that it makes debugging very much simpler (*e.g.* the `MemPool` (Sec.52) object inside `RingZZImpl` will be destroyed while the input and output streams are still functioning, thus allowing the `MemPool` (Sec.52) destructor to report any anomalies). And of course, it is simply good manners to clean up properly at the end of the program.

37.3 Maintainer Documentation

To implement the restriction that only one `GlobalManager` may exist at any one time, the first instruction in the ctor checks that the global variable `GlobalManager::ourGlobalDataPtr` is null. If it is null, it is immediately set to point the object being constructed. At the moment, this check is not threadsafe.

The ctor for `GlobalManager` is fairly delicate: *e.g.* the functions it calls cannot use the functions `RingZZ()` and `RingQQ()` since they will not work before the `GlobalManager` is registered.

The two functions `MakeUniqueCopyOfRingZZ` and `MakeUniqueCopyOfRingQQ` are supposed to be accessible only to the ctor of `GlobalManager`; they create the unique copies of those two rings which will be stored in the global data. The functions are defined in `RingZZ.C` and `RingQQ.C` respectively but do not appear in the corresponding header files (thus making them "invisible" to other users).

The dtor for `GlobalManager` checks that `RingZZ` and `RingQQ` are not referred to by any other values (*e.g.* ring elements which have been stored in global variables). A rude message is printed on `cerr` if the reference counts are too high, and a program crash is likely once the `GlobalManager` has been destroyed.

37.3.1 GMPMemMgr

The `GMPMemMgr` class performs the necessary steps for setting the memory manager for GMP values. At the moment there are essentially two choices: use the system memory manager, or use a `MemPool` (Sec.52) to handle the memory for small values. The first parameter to the ctor for `GMPMemMgr` says which sort of memory manager to use. If the system allocator is chosen, then the ctor does nothing (since the GMP default is the system manager); similarly nothing is done when the `GMPMemMgr` object is destroyed. The second argument is completely ignored when the system allocator is chosen.

The situation is more complicated if CoCoALib's custom allocator is to be used. The second argument specifies the *slice size* (in bytes) which is to be used – the implementation may automatically increase this value to the next convenient value (see also the documentation for `MemPool` (Sec.52)). The slice size defines what a GMP *small value*

is: it is a value whose GMP internal representation fits into a single slice. The memory for small values is managed by a (global) `MemPool`, while the memory for larger values is managed by the standard `malloc` family of functions.

Since the only place a `GMPMemMgr` object appears is as a data field in a `GlobalManager`, we have an automatic guarantee that there will be at most one `GMPMemMgr` object in existence – this fact is exploited (implicitly) in the ctor and dtor for `GMPMemMgr` when calling the GMP functions for setting the memory management functions.

Of the `alloc/free/realloc` functions which are handed to GMP, only `CoCoA_GMP_realloc` displays any complication. GMP limbs can be stored either in memory supplied by the `MemPool` belonging to a `GMPAllocator` object or in system allocated memory; a reallocation could cause the limbs to be moved from one sort of memory to the other.

37.3.2 GlobalSettings

The `GlobalSettings` class serves only to allow a convenient syntax for specifying the parameters to the `GlobalManager` ctor. The only mild complication is the `operator+` for combining the ctor parameters, where we must check that nonsensical or ambiguous combinations are not built.

37.4 Bugs, Shortcomings, etc

2010-09-30 The private copies of `RingZZ` and `RingQQ` are now direct members, previously they were owned via `auto_ptr`s. The new implementation feels cleaner, but has to include the definitions of `ring` and `FractionField` in the header file.

You cannot print out a `GlobalManager` object; is this really a bug?

Ctor for `GlobalManager` is **NOT THREADSAFE**.

Should the ctor for `GlobalManager` set the globals which control debugging and verbosity in `MemPool` (Sec.52)s?

38 hilbert (Anna Bigatti)

38.1 hilbert

I'm just using the C code I wrote for CoCoA-4. It will be entirely rewritten in C++

The only usable function is

```
RingElem HilbertNumeratorMod(PolyRing HSRing, ideal I);
```

but it works ONLY if I is a monomial ideal. HSRing is the ring where the Hilbert Series Numerator should live.

39 ideal (John Abbott)

39.1 Examples

- ex-RingHom3.C
- ex-AlexanderDual.C
- ex-QuotientBasis.C

39.2 User documentation

The class `ideal` is for representing values which are ideals of some `ring` (Sec.69). There are several ways to create an `ideal`:

NOTE: THIS SYNTAX WILL PROBABLY CHANGE

- `ideal I(r)` – `I` is the principal ideal generated by `r` (a `RingElem` (Sec.71)) in the ring `owner(r)`
- `ideal I(r1, r2)` – `RingElem` (Sec.71)s in the same `ring` (Sec.69)

- `ideal I(r1, r2, r3)` – `RingElem` (Sec.71)s in the same `ring` (Sec.69)
- `ideal I(r1, r2, r3, r4)` – `RingElem` (Sec.71)s in the same `ring` (Sec.69)
- `ideal I(R, gens)` – `I` is the ideal of `R` generated by the `RingElem` (Sec.71)s in the C++ vector<> `gens`, (all in the same `ring` (Sec.69))
- `ideal I(gens)` – if `gens`=[] throws ERROR, otherwise equivalent to `I(owner(gens[0]), gens)`

  If you want to make an ideal in `R` with no generators use this syntax

```
ideal(R, vector<RingElem>())
```

39.2.1 Operations

The permitted operations on `ideal`s are: let `I` and `J` be two ideals of the same ring

- `I+J` – the sum of two ideals
- `I += J` – equivalent to `I = I+J`
- `intersection(I, J)` – intersection of two ideals
- `colon(I, J)` – the quotient of two ideals

  We may also enquire about certain properties of an ideal:
- `IsZero(I)` – true iff the ideal is a zero ideal
- `IsOne(I)` – true iff the ideal is the whole ring
- `IsMaximal(I)` – true iff the ideal is maximal in its ring (i.e. iff the quotient ring is a field)
- `IsPrime(I)` – true iff the ideal is prime (i.e. quotient ring has no zero-divisors)
- `IsContained(I, J)` – true iff the ideal `I` is a subset of the ideal `J`
- `IsElem(r, I)` – true iff `r` is an element of the ideal `I`
- `I == J` – true iff the ideals are equal (their generating sets may be different)
- `AmbientRing(I)` – the ring in which the ideal `I` resides
- `gens(I)` – a C++ vector<> of `RingElem` (Sec.71)s which generate `I`
- `TidyGens(I)` – a C++ vector<> of `RingElem` (Sec.71)s which generate `I` (this generating set is in some way "reduced", and will never contain a zero element)

Queries

It is also possible to give some information about an ideal:

```
I->UserAssertsIsPrime()       to specify that ``I'' is known to be prime
I->UserAssertsIsNotPrime()    to specify that ``I'' is known not to be prime
I->UserAssertsIsMaximal()     to specify that ``I'' is known to be maximal
I->UserAssertsIsNotMaximal()  to specify that ``I'' is known not to be maximal
```

Making an incorrect assertion using these functions may lead to a program crash, or at least to poorer run-time performance.

39.2.2 Functions for ideals in polynomial rings

Additional functions for an ideal `I` in a `SparsePolyRing` (Sec.87) `P`.

- `IsZeroDim(I)` – true iff `I` is zero-dimensional
- `IsHomog(I)` – true iff `I` is homogeneous
- `AreGensMonomial(I)` – true iff given `gens(I)` are all monomial. NB `0` is NOT monomial
- `AreGensSquareFreeMonomial(I)` – true iff given `gens(I)` are all monomial and radical. NB `0` is NOT monomial
- `GBasis(I)` – same as `TidyGens` (stored into `I` for future use)
- `LT(I)` – leading term ideal (also knows as *initial ideal*)
- `homog(h, I)` – homogenization with the indeterminate `h`, a `RingElem` (Sec.71), indeterminate in `AmbientRing(I)`
- `QuotientBasis(I)` – basis of the quotient as a `K`-vector space
- `PrimaryDecomposition(I)` – only for square free monomial ideals (for now)

Monomial ideals

Additional functions for a monomial ideal `I` in a `SparsePolyRing` (Sec.87)

- `PrimaryDecompositionMonId(I)` – only for square free monomial ideals (for now)
- `AlexanderDual(I)` – only for square free monomial ideals (for now)

Using Frobby library

- `PrimaryDecompositionFrobby(I)`
- `AlexanderDual(I)`, `AlexanderDual(I, pp)`
- and more...

39.2.3 Writing new types of ideal

Anyone who writes a new type of ring class will have to consider writing a new type of ideal class to go with that ring. The ideal class must be derived from the abstract class IdealBase (and to be instantiable must offer implementations of all pure virtual functions). Be especially careful to update the data members IamPrime and IamMaximal in the non-const member functions (add, intersection, and colon).

Some guidance may be obtained from looking at the FieldIdealImpl class which implements ideals in a field (there are only two: ideal(0) and ideal(1)). See the file FieldIdeal.txt

39.3 Maintainer documentation for the classes ideal, IdealBase

The class ideal is little more than a reference counting smart pointer class pointing to an object of type derived from IdealBase. This approach allows many different implementations of ideals to be manipulated in a convenient and transparent manner using a common abstract interface.

The abstract class IdealBase specifies the interface which every concrete ideal class must offer. It is more complicated than one might expect partly because we want to allow the advanced user to tell the ideal whether it has certain important properties (which might be computationally expensive to determine automatically).

39.4 Bugs, Shortcomings and other ideas

The maintainer documentation is still quite incomplete.

Shouldn't ideals be created by a function called NewIdeal???

I am not at all sure about the wisdom of having implemented `IamPrime` and `IamMaximal`. It seems to be terribly easy to forget to update these values when ideal values are modified (e.g. in IdealBase::add). It has also led to rather more complication that I would have liked. BUT I don't see how to allow the user to state that an ideal is maximal/prime without incurring such complication.

Functions to examine the bool3 flags could be handy for *heuristic* short-cuts when an ideal is already known to have a certain property.

Is it worth having a constructor for principal ideals generated by a number rather than a RingElem? e.g. NewIdeal(R,5) or NewIdeal(R,BigInt(5)).

Several member functions have names not in accordance with the coding conventions.

40 empty (John Abbott)

40.1 Examples

- ex-empty.C

40.2 User documentation

The functions here are for computing generators of the vanishing ideal of a set of points (*i.e.* all polynomials which vanish at all of the points).

The functions expect two parameters: a polynomial ring `P`, and a set of points `pts`. The coordinates of the points must reside in the coefficient ring of `P`. The points are represented as a matrix: each point corresponds to a row.

40.2.1 Operations

The main functions available are:

- `IdealOfPoints(P,pts)` computes the vanishing ideal in `P` of the points `pts`.
- `BM(P,pts)` computes the reduced Groebner basis of the vanishing ideal in `P` of the points `pts`;

40.3 Maintainer documentation

Impl is simple/clean rather than fast.

There was a minor complication to handle the case where the dim of the space in which the points live is less than the number of indets in the polyring.

40.4 Bugs, shortcomings and other ideas

2013-01-21 there is only a generic impl (which is simple but inefficient).

The name `BM` is too short?

40.5 Main changes

2013

- January (v0.9953): first release

41 IntOperations (John Abbott)

41.1 User documentation

The class `BigInt` (Sec.8) is intended to represent (signed) integers of practically unlimited range; it is currently based on the implementation in the GMP *big integer* library.

The class `MachineInt` (Sec.46) is intended to help you write functions which accept arguments whose type is a machine integer offering a safe interface for signed and unsigned maching integers.

When the CoCoALib documentation says *integer* it means both big integers and machine integers which are mainly interchangable. Do remember, though, that operations between two machine integers are handled directly by C++, and problems of overflow can occur.

The few exceptions accepting only `long` are clearly indicated and usually apply to indices and subscripts.

41.1.1 Examples

- ex-BigInt1.C
- ex-BigInt2.C
- ex-BigInt3.C

41.1.2 Queries

- `IsEven(n)` – true iff `n` is even
- `IsOdd(n)` – true iff `n` is odd
- `IsDivisible(n,d)` – true iff `n` is divisibile by `d`
- `IsSquare(n)` – true iff `n` is a perfect square
- `IsPower(n)` – true iff `n` is a perfect k-th power for some k > 1
- `IsExactIroot(X,N,r)` – true iff `N` is a perfect `r`-th power, assigns `iroot(N,r)` to `X`

Only for `BigInt` (Sec.8)

- `IsZero(n)` – true iff `n` is zero
- `IsOne(n)` – true iff `n` is 1
- `IsMinusOne(n)` – true iff `n` is -1

41.1.3 Operations

Infix operators

1. normal arithmetic (potentially inefficient because of temporaries)
   - `=` assignment
   - `+` the sum
   - `-` the difference
   - `*` the product
   - `/` integer quotient (**truncated** "towards zero")
   - `%` remainder, satisfies a = b*(a/b)+(a%b); see also `LeastNNegRemainder` and `SymmRemainder`
2. arithmetic and assignment
   - `+=`, `-=`, `*=`, `/=`, `%=` – definitions as expected; if RHS is a `BigInt` (Sec.8) LHS must be `BigInt` (Sec.8)
3. arithmetic ordering

- `==`, `!=`
- `<`, `<=`, `>`, `>=` – comparison (using the normal arithmetic ordering) – see also the `cmp` function below.

4. increment/decrement
   - `++`, `--` (prefix, *e.g.* `++a`) use these if you can
   - `++`, `--` (postfix, *e.g.* `a++`) avoid these if you can, as they create temporaries

cmp

(three way comparison)

- `cmp(a, b)` – returns an `int` which is `< 0`, `== 0`, or `> 0` if `a < b`, `a == b`, or `a > b` respectively
- `CmpAbs(a,b)` – same as `cmp(abs(a),abs(b))` but might be faster.

Sundry standard functions

(Several basic number theoretical operations are defined in `NumTheory` (Sec.55)) Let `n` be an integer, let `hi`, `lo` be machine integers

- `abs(n)` – the absolute value of `n`
- `sign(n)` – result is `int`: -1 if `n<0`, 0 if `n==0`, and +1 if `n>0`
- `LeastNNegRemainder(x,m)` – least non-negative remainder; throws `ERR::DivByZero` if `m==0`
- `SymmRemainder(x,m)` – symmetric remainder; throws `ERR::DivByZero` if `m==0`
- `log(n)` – natural logarithm of the absolute value of `n` (as a `double`)
- `RoundDiv(N,D)`– rounded division of `N` by `D`, halves round towards +infinity
- `isqrt(N)` – the (truncated) integer part of the square root of `N`
- `ILogBase(N,b)`– the integer part of log(abs(N))/log(b); error if N=0 or b<2 (as a `long`)

These functions return `BigInt` (Sec.8)

- `power(a, b)` – returns `a` to the power `b` (`b` must be >=0)
- `SmallPower(a, b)` – returns `a` to the power `b` (`b` must be >=0) assuming result fits into a `long`
- `factorial(n)` – factorial for non-negative `n`
- `LogFactorial(n)` – approx natural log of factorial (abs.err. < 5*10^(-8))
- `RangeFactorial(lo,hi)` – `lo*(lo+1)*(lo+2)*...*hi`
- `binomial(n, r)` – binomial coefficient
- `fibonacci(n)` – `n`-th Fibonacci number
- `iroot(N,r)` – the (truncated) integer part of the `r`-th root of `N` (see also `IsExactIroot`)
- `RandomBigInt(lo, hi)` – a uniform random integer `N` s.t. `lo <= N <= hi` (see `random` (Sec.??) for details).

Conversion functions

Only for `BigInt` (Sec.8)

- `mantissa(n)` – n represented as a floating-point number. if `n` is zero, produces 0.0 otherwise if `n>0` a value between 0.5 and 0.999... otherwise (when `n<0`) a value between -0.5 and -0.999... The bits of the floating point result are the topmost bits of the binary representation of `n`.
- `exponent(n)` – result is a `long` whose value is the least integer e such that 2^e > abs(n). If `n` is zero, result is zero.

Miscellany

Only for `BigInt` (Sec.8)

- `NumDigits(n, b)` – the number of digits (as a `long`) `n` has when written in base `b` (the result may sometimes to be too large by 1)

Procedures for arithmetic

Assignment is always to leftmost argument(s) `a`, a `BigInt` (Sec.8). Second and/or third argument of type `BigInt` (Sec.8).

- `add(a, b, c)` – a = b+c
- `sub(a, b, c)` – a = b-c
- `mul(a, b, c)` – a = b*c
- `div(a, b, c)` – a = b/c (truncated integer quotient)
- `mod(a, b, c)` – a = b%c (remainder s.t. b = quot*c + rem)
- `quorem(a, b, c, d)` – same as a = c/d, b = c%d
- `divexact(a, b, c)` – a = b/c (fast, but division must be exact)
- `power(a, b, c)` – a = b^c
- `neg(a, b)` – a = -b
- `abs(a, b)` – a = abs(b)

41.1.4 Error Conditions and Exceptions

Error conditions are signalled by exceptions. Examples of error conditions are impossible arithmetic operations such as division by zero, overly large arguments (*e.g.* second argument to binomial must fit into a machine `long`), and exhaustion of resources.

Currently the exception structure is very simplistic:

- exceptions indicating exhaustion of resources are those from the system, this library does not catch them;
- all other errors produce a `CoCoA::ErrorInfo` exception; for instance

`ERR::ArgTooBig`	value supplied is too large for the answer to be computed
`ERR::BadArg`	unsuitable arg(s) supplied (or input number too large)
`ERR::BadNumBase`	the base must be between 2 and 36
`ERR::BadPwrZero`	attempt to raise 0 to non-positive power
`ERR::DivByZero`	division by zero
`ERR::ExpTooBig`	exponent is too large
`ERR::IntDivByNeg`	inexact integer division by a negative divisor
`ERR::NegExp`	negative exponent
`ERR::ZeroModulus`	the modulus specified is zero

41.2 Maintainer Documentation

The implementation of `cmp` is more convoluted than I'd like; it must avoid internal overflow.

The implementation of `RoundDiv` was more difficult than I had expected. Part of the problem was making sure that needless overflow would never occur: this was especially relevant in the auxiliary functions `uround_half_up` and `uround_half_down`. It would be nice if a neater implementation could be achieved – it seems strange that the C/C++ standard libraries do not already offer such a function. The standard C functions `lround` almost achieves what is needed here, but there are two significant shortcomings: rounding is always away from zero (rather than towards +infinity), and there could be loss of accuracy if the quotient exceeds 1/epsilon. There is also a standard function `ldiv` which computes quotient and remainder, but it seems to be faster to compute the two values explicitly.

41.3 Bugs, shortcomings and other ideas

The power functions could allow high powers of -1,0,1 (without complaining about the exponent being too big).

Only partial access to all the various division functions offered by the C interface to GMP. Many other GMP functions are not directly accessible.

`IsExactIroot` has rather a lot of signatures.

41.4 Main changes

2012

- May (v0.9951):
  - moved common operations on `BigInt` (Sec.8) and `MachineInt` (Sec.46) together into `IntOperations` -

42 io (John Abbott)

42.1 User Documentation for files io.H and io.C

These files supply four standard global I/O channels for the CoCoA library. These channels mimic the global channels present in the C++ STL: `cin`, `cout`, `cerr`, and `clog`.

The current choices for these four channels can be obtained by calling these functions (inside namespace CoCoA):

```
std::istream& GlobalInput();
std::ostream& GlobalOutput();
std::ostream& GlobalErrput();
std::ostream& GlobalLogput();
```

By default the standard global I/O channels for the CoCoA library are the corresponding ones of the standard C++ library. Alternative choices may be specified by calling these functions (see *NOTE*)

```
std::istream& SetGlobalInput(std::istream& in);
std::ostream& SetGlobalOutput(std::ostream& out);
std::ostream& SetGlobalErrput(std::ostream& err);
std::ostream& SetGlobalLogput(std::ostream& log);
```

In each case the value returned is the previous istream/ostream associated with that channel.

NOTE the procedures for changing the settings of the global i/o streams maintain a reference to the value supplied. If you use a local variable as argument, make sure that its value is not destroyed before you have finished i/o on it. See the example program ex-io.C.

```
void InputFailCheck(const std::istream& in, const char* const FnName);
```

This function simply checks that the stream `in` is in a good state; if not, it throws an `ERR::InputFail` indicating the function name `FnName`.

42.2 Maintenance notes for the files io.H and io.C

The implementation could hardly be simpler.

As recommended in Meyers's book I have put the globals in an anonymous namespace. I chose to use plain pointers for the global variables `InPtr`, `OutPtr`, `ErrPtr` and `LogPtr`; references are unsuitable because they cannot be reseated in C++, and `auto_ptr` is unsuitable because we do not want to own the streams. I do not believe that there can be problems with race-conditions when these four global variables are initialized since we use only the addresses of `std::cin`, etc. However, there could be race-condition problems with subsequent changes to the values.

42.3 Bugs, Shortcomings, etc

The names `GlobalInput` etc are rather cumbersome. It is also annoying to have to call them as functions. I could use automatic type conversions to eliminate the need for the "()"s – I'll wait until people start complaining. Suggestions for better names are welcome.

Another reason for having a new class to represent `GlobalOutput` etc. is that the template for printing `std::lists` and `std::vectors` could easily clash with a user's definition of `operator<<`.

It is tempting to make the simple functions in io.C into inline functions, but inlining is tricky with the global variables in an anonymous namespace. And anyway the minor gain in performance will easily be swamped by the high costs of actually conducting I/O; so making them inline would really be quite pointless.

It would be nice to avoid the potential pitfalls of dangling references, but I do not currently see how to achieve this.

43 JBMill (Mario Albert)

43.1 User documentation for Janet Basis

The files `JBDatastructure.H`, `JBSets.H` and `JBEnv` intodruces several classes for computing and working with **Janet Basis**. A normal user of the CoCoa library will use only the class `JBMill` (Sec.43). With this class the user can do anything, which is related to Janet Bases. Starting with the computation of the Janet Basis for degree compatible orderings up to computing e.g. extremal betti numbers.

43.1.1 Computing a Janet Basis

There are several ways to compute a Jane Basis: In the following let `gens` a C++ vector of `RingElem` (Sec.71)s (all elements in gens must be in the same ring), `I` an ideal, `crits` a C++ bitset<3>, `output` a flag, which specifies the Output (allowed flags are `GB` or `JB` (default is `JB`)) and `flag` a flag, which specifies the used algorithm (allowed flags are `TQDegree`, `TQBlockHigh`, `TQBlockLow` (default is `TQBlockLow`)). `crits` specifies the used involutive criteria (by default there are all three activated). `gens` or `I` contains the input set.

The following algorithms only compute a Groeber Basis (resp. Janet Basis). The output is always a C++ vector of `RingElem` (Sec.71)s.

- `JanetBasis(gens, crits, output, strategy)` – `crits`, `output` and `strategy` are not necessary. If they are not specified the algorithm uses the default values.
- `JanetBasis(I, crits, output, strategy)` – `crits`, `output` and `strategy` are not necessary. If they are not specified the algorithm uses the default values.

  The following algorithms returns a `JBMill`
- `ExtendedJanetBasis(gens, crits, output, strategy)` – `crits`, `output` and `strategy` are not necessary. If they are not specified the algorithm uses the default values.
- `ExtendedJanetBasis(I, crits, output, strategy)` – `crits`, `output` and `strategy` are not necessary. If they are not specified the algorithm uses the default values.

43.1.2 Using the JBMill

Always Works

In the following let `mill` be a `JBMill`. The Janet Basis contained in `mill` generate the Ideal I which is a subset of the PolyRing P.

JBReturnJB(mill)	returns the Janet Basis of mill as C++ vector of RingElem (Sec.71)s
JBReturnGB(mill)	returns the Groebner Basis of mill as C++ vector of RingElem (Sec.71)s
JBIsPommaret(mill)	Returns true if the Janet Basis of I is also a Pommaret Basis, otherwise fals
JBIsHomogenous(mill)	Returns true if the Janet Basis of I is homogenous, otherwise false
JBIsMonomialIdeal(mill)	Returns true if the Janet Basis of I is a monomial ideal, otherwise false
JBOutputMultVar(mill)	Prints the Janet-multiplicative variables of the Janet Basis of I
JBOutputNonMultVar(mill)	Prints the Janet-nonmultiplicative variables of the Janet Basis of I
JBMultVar(mill)	Returns the Janet-multiplicative variables of the elements in the Janet-Basis a
JBNonMultVar(mill)	Returns the Janet-nonmultiplicative variables of the elements in the Janet-Bas
JBStandardRepresentation(mill, f)	f must be in the same ring as mill. This function computes the involutive sta
JBOutputStandardRepresentation(mill, f)	Same as above. But only prints the result.
JBNormalForm(mill, f)	f must be in the same ring as mill. This function computes the involutive no
JBHilbertPol(mill, s)	returns the Hilbert Polynomial of P/I with variable s, which must be a RingE
JBHilbertFunc(mill, number)	return the value of the Hilbert Function of P/I at the position number. numbe
JBHilbertFunc(mill)	print the Hilbert Function of P/I
JBHilbertSeries(mill, s)	returns the Hilbert Series of P/I in terms of s. s must be a RingElem (Sec.71)
JBSyzygy(mill)	Computes the first Syzygy of I. It returns a FGModule
JBDim(mill)	Computes the dimension of P/I. It returns a number of type long
JBCls(mill, f)	It computes the class of LPP(f) if f is a RingElem (Sec.71). If f is a PPMonoi
JBMinCls(mill, f)	It computes the minimal class of the Janet Basis of I. It returns a number of t
JBElementsWithClass(mill, n)	It retuns all elements of the Janet Basis of class n as a C++ vector

The Basis must be monomial

JBComplementaryDecomposition(mill)	Returns the complementary decomposition of I as C++ vector<pair<PPMonoidEl
JBStandardPairs(mill)	Returns the standard Pairs of I as C++ vector<pair<PPMonoidElem, vector<bo

The Basis must be Pommaret

JBMaxStronglyIndependentSet(mill)	Returns a maximal strongly independent set modulo I as a C++ vector
JBDegPommaretClass(mill, n)	Returns the maximal degree of elements with class n in the Janet Basis as number of

The Basis must be a Pommaret and Homogenous

JBDepth(mill)	Computes the depth of I. It returns a number of type long
JBProjDim(mill)	Computes the projective dimension of I. It returns a number of type long
JBIsCohenMacaulay(mill)	Returns true if I is a Cohen-Macauay ring, otherwise false
JBRegularSequence(mill)	Returns a maximal requlars sequence of I as a C++ vector

The Basis must be a Pommaret and Homogenous and the ordering must be degrevlex

JBRegularity(mill)	Returns the regularity of I as number of type long
JBCastelnuovoMumfordRegularityI](mill)	Same as above|
JBSaturation(mill)	Returns the generating set of the Saturation of I as C++ vector|
JBSatiety(mill)	Returns the satiety of I as natural number of type long. If the ideal is saturate
JBExtremalBettiNumbers(mill)	Returns the extremal betti numbers of the ideal I as C++ map<pair<long, l

The Basis must be a Pommaret and Homogenous and the ordering must be degrevlex and the ideal must be CohenMacaulay

JBSocle(mill)	Returns the generating set of the socle of I as C++ vector

43.1.3 Examples

- ex-Janet.C
- ex-Janet2.C
- ex-Janet3.C

43.2 Maintainer documentation for JBDatastructure.C, JBSets.C, JBEnv.C

We only explain the basic structure because there is very much code (~5500 loc). The implementation is divided in three parts:

43.2.1 JBDatastructure.C

Here we define the basic datastructures, which are necessary to compute Janet-Bases fast.

JanetTriple

These class contains three informations. First of all it contains the polynomial of our generating set. The second part is the *ancestor*. Normally the ancestor is the leading monomial of the polynomial contained in the Triple. But if the polynomial is the result of an involutive prolongation the ancestor is the leading monomial of the origin polynomial. The last part contains a C++ `vector<bool>` which shows with which variables we already prolonged.

JanetTree

The most important datastructure, but maybe also the worst one... The `JanetTree` is a binary tree where we order the generating set in variable and degree direction. For further information you have to look at the literatur. The `JanetTree` is implemented as a nested set of C++ vectors (I am not sure if this was a right decision, but I guess to change this could be even worse, than living with it...) The `JanetTree` contains only in the leafs a reference to the JaneTriples. The nodes contain a Handle class, which are either an internal handle class (no ``JanetTriple'') or a leaf class (contain only a triple) Apart from that the nodes only contains datas about their position.

43.2.2 JBSets.C

The algorithms for computing Janet-Bases dealing mainly with to sets `T` and `Q` (and `P`), which contain `JanetTriple`. In every iteration some elements goes from `T` to `Q` and vica versa. In the first implementation we discoverd that it is really expensive to delete and element in `T` and copy it to `Q`. Therefore we decide to intodruce a new Set `BasicSet`. In this set every `JanetTriple` is included. The Sets `T`, `Q` and `P` contain only Pointers to this `BasicSet`. `BasicSet` is only a C++ list (because inserting and deleting of elements don't change other iterators). `T`, `Q`, `P` are C++ multisets. For this multiset we defined in class `JBSets` a inner class `CompareIterator`, that we need for odering elements in `T`, `Q` and `P`.

In addition this class contains some useful functions to deal and manipulating elements in `T`, `Q` and `P`.

43.2.3 JBEnv.C

This is the *main file*.

JBEnv

The class `JBEnv` contains the basic informations about the ring.

JBFlag

This class stores informations about the options which we use in the computation

In short: Everything else! The class contains the main algorithms for computing Janet Basis (`DegreeTQ` and `BlockTQ`). And every function which deals with the JanetBase. Maybe this is not a wise decision because this class gets very big. For the implemented algorithms I refer to the literature...

43.3 Bugs, Shortcomings and other ideas

TODO Index.html in examples **The code requires tests! Immediately!** we always assume that `x1 > x2 > x3 > ... > xn`

44 leak-checker (John Abbott)

44.1 User documentation

leak_checker is a standalone program included with the distribution of the CoCoA library. It can help track down memory leaks. If you have never used leak_checker before, it may be helpful to try the small example given in the file debug_new.txt.

This program scans output produced by a program run either with the debugging versions of `new/delete` (see `debug_new` (Sec.15)) or using `MemPool` (Sec.52)s with debugging level set high enough that each allocation/deallocation produces a verbose report (see `MemPool` (Sec.52)). `leak_checker` pairs up every `free` message with its corresponding `alloc` message, and highlights those `alloc` messages which do not have a corresponding `free` message. In this way probable memory leaks can be tracked down.

To use leak_checker with the debugging version of global new/delete, see the file `debug_new` (Sec.15) (which includes a small example to try). To use leak_checker with MemPools, you must compile with the CPP flag CoCoA_MEMPOOL_DEBUG set – this probably entails recompiling all your code; see `MemPool` (Sec.52) for details. In either case, with debugging active your program will run rather more slowly than usual, and will probably produce large amounts of output detailing every single allocation/deallocation of memory – for this reason it is best to use smaller test cases if you can. Put the output into a file, say `memchk`.

Now, executing `leak_checker memchk` will print out a summary of how many alloc/free messages were found, and how many unpaired ones were found; beware that leak_checker may take a long time if your program's output details many allocations and deallocations. The file `memchk` will be modified if unpaired alloc/free messages were found: an exclamation mark is placed immediately after the word ALLOC (where previously there was a space), thus a search through the file memchk for the string `ALLOC!` will find all unpaired allocation messages.

Each allocation message includes a sequence number (`seq=...`). This information can be used when debugging. For instance, if the program leak_checker marks an unpaired allocation with sequence number 500 then a debugger can be used to interrupt the program the 500th time the allocation function is called (the relevant function is either debug_new::msg_alloc or CoCoA::MemPool::alloc). Examining the running program's stack should fairly quickly identify precisely who requested the memory that was never returned. Obviously, to use the debugger it is necessary to compile your program with the debugger option set: with gcc this option corresponds to the flag `-g`.

WARNING: debug_new handles ALL new/delete requests including those arising from the initialization of static variables within functions (and also those arising from within the system libraries). The leak_checker program will mark these as unfreed blocks because they are freed only after main has exited (and so cannot be tracked by debug_new).

44.2 Maintainer documentation

This was formerly a C program (as should be patently evident from the source code). It requires a file name as input, and then scans that file for messages of the form

```
ALLOC 0x....
FREED 0x....
```

(such messages are produced by the global operators new/delete in debug_new.C and also by the verbose version of MemPool (with debug level $>= 3$)) It then attempts to match up pointer values between ALLOC and FREED messages. Finally the file is scanned again, and any ALLOC or FREED messages which were not matched up are modified by adding an exclamation mark (!) immediately after the word ALLOC or FREED.

The matching process is relatively simplistic. During an initial scan of the file all ALLOC/FREED messages are noted in two arrays: one indicating the type of message, the other containing the pointer value. Initially the two types are UNMATCHED_ALLOC and UNMATCHED_FREE, as the matching process proceeds some of these will become MATCHED_ALLOC or MATCHED_FREE (accordingly); obviously the types are changed in pairs.

The matching process merely searches sequentially (from the first entry to the last) for pointer values of type UNMATCHED_FREE. For each such value it then searches back towards the first entry looking for an UNMATCHED_ALLOC with the same pointer value. If one is found, then both types are switched to MATCHED_xxx. If none is found, the UNMATCHED_FREE is left as such. The main loop then resumes the search for the next UNMATCHED_FREE to try to pair up. This approach does get slow when there are very many ALLOC/FREED messages, but I do not see any simple way of making it faster.

44.3 Bugs, shortcomings, and other ideas

This program gets painfully slow on large files. It is also rather crude, though quite effective at its job.

45 library (Anna Bigatti)

45.1 User documentation for file library.H

library.H is generated by running `make` in the `include/CoCoA/` directory (which is also called by the general `make` in the CoCoALib directory).

It includes all the .H files of CoCoALib, so, copying the lines

```
#include "CoCoA/library.H"
using namespace CoCoA;
```

is the easiest way to use it (see the examples directory)

When you include `library.H` you are also guaranteed to include

```
#include <algorithm>// using std::transform; from apply.H
#include <bitset>   // using std::bitset;    from DivMask.H
#include <cstddef>  // using std::size_t;    from MemPool.H and BigInt.H
#include <exception>// using std::exception; from error.H
#include <gmp.h>    //                       from BigInt.H
#include <iosfwd>   // using std::ostream;   from PPOrdering.H and BigInt.H
#include <iostream> // using std::istream; using std::ostream;  from io.H
#include <list>     // using std::list;      from QBGenerator.H and io.H
#include <memory>   // using std::auto_ptr;  from MemPool.H
#include <string>   // using std::string;    from MemPool.H and symbol.H
#include <vector>   // using std::vector;    from DenseMatrix.H and io.H
```

For maintenance purposes we list the most stable files including them. This list is probably not complete, but should be pretty reliable in the years to come.

45.2 Common includes

To ensure portability you should specify what you use and where it is defined. Moreover you should not have a `using` in a `.H` file.

Here is a list of the most common includes, for more details look at Jossutis *C++ Standard Library.*

```
#include <algorithm>
using std::back_inserter;
using std::copy;
using std::count_if;
using std::fill;
using std::find;
using std::find_if;
```

```
using std::for_each;
using std::max;
using std::min;
using std::sort;
using std::stable_sort;
using std::swap;

#include <list>
using std::list;

#include <cstddef>
using std::size_t;

#include <cstdlib>
using std::abs;
using std::size_t;

#include <cstring>
using std::memcpy;

#include <functional>
using std::binary_function;
using std::bind1st;
using std::bind2nd;
using std::less;
using std::mem_fun_ref; // for calling GPair::complete on GPairList

#include <iostream>
using std::endl;
using std::flush;
using std::ostream;

#include <iterator>

#include <limits>
using std::numeric_limits;

#include <memory>
using std::auto_ptr;

#include <new>
// for placement new

#include <set>
using std::set;

#include <string>
using std::string;

#include <utility>
using std::make_pair;

#include <vector>
using std::vector;
```

46 MachineInt (John Abbott)

46.1 User documentation for MachineInt

The class `MachineInt` is intended to help you write functions which accept arguments whose type is a machine integer (see **Why?** below). We recommend that you use `MachineInt` only to specify function argument types; other uses may result in disappointing performance.

You cannot perform arithmetic directly with values of type `MachineInt`. The primary operations are those for extracting a usable value from a `MachineInt` object:

46.1.1 Operations

Arithmetic directly with `MachineInt` values is not possible. The value(s) must be converted to `long` or `unsigned long` before operating on them.

46.1.2 Queries and views

- `IsNegative(N)` – true iff `N` is negative, if false the value can be extracted as an `unsigned long`, if true the value can be extracted as a signed `long`
- `IsSignedLong(N)` – true iff `N` can be extracted as a signed `long`
- `AsUnsignedLong(N)` – extract `N` as an `unsigned long` – see NOTE!
- `AsSignedLong(N)` – extract `N` as a signed `long` – see NOTE!
- `IsInRange(lo,x,hi)` – true iff `lo <= x <= hi`

46.1.3 NOTE: converting to long or unsigned long

You should not call `AsUnsignedLong` if the value is negative, nor should you call `AsSignedLong` if the value is large and positive — currently, an error is signalled only if debugging is active. Here's an outline of the recommended usage:

```
void SomeProcedure(const MachineInt& N)
{
  if (IsNegative(N))
  {
    long n = AsSignedLong(N);
    ...
  }
  else // N is non-negative
  {
    unsigned long n = AsUnsignedLong(N);
    ...
  }
}
```

46.1.4 Why?

The class `MachineInt` was created in an attempt to circumvent C++'s innate automatic conversions between the various integral types; most particularly the silent conversion of negative signed values into unsigned ones.

Various C++ programming style guides recommend avoiding unsigned integer types. Unfortunately values of such types appear frequently as the result of various counting functions in the STL. So it is somewhat impractical to avoid unsigned values completely.

The class `MachineInt` employs automatic user-defined conversions to force all integral values into the largest integral type, *viz.* `long` or `unsigned long`. An extra "sign bit" inside a `MachineInt` indicates whether the value is negative (*i.e.* must be regarded as a signed `long`).

Passing an argument as a `MachineInt` is surely not as fast as using a built in integral type, but should avoid "nasty surprises" which can arise with C+'s automatic conversions (*e.g.* a large `unsigned long` could be viewed as a negative `long`).

46.2 Maintainer documentation for MachineInt

On the whole everything is very simple; the hard part was establishing a reasonable design that interoperates with C++'s overload resolution rules.

An object of type `MachineInt` contains two data fields:

- `myValue` – the original integer value converted to `unsigned long`
- `IamNegative` – true iff the original value was (signed and) negative

The flag `IamNegative` allows the field `myValue` to be interpreted correctly: if `IamNegative` is `true` then the correct value of `myValue` may be obtained by casting it to a (signed) `long`; conversely, if `IamNegative` is `false` then the value of `myValue` is correct as it stands (*i.e.* as an `unsigned long`).

Most functions are so simple that an inline implementation is appropriate.

The implementation of the function `abs` will work correctly even if the value being represented is the most negative signed `long`. Note that the C++ standard allows the system to produce an error when negating a `long` whose value is the most negative representable value; in contrast, operations on `unsigned long` values will never produce errors (except division by zero).

The impl of `IsInRange` is a bit involved; it must avoid overflow, and may not assume anything about the internal representations of signed and unsigned long values.

46.3 Bugs, Shortcomings and other ideas

My biggest doubt is whether this is really the right way to tackle the problem of silent automatic conversion between `long` and `unsigned long`. Anyway, I'm using it (until a better solution comes along).

46.4 Main changes

2011

- November (v0.9949): this class was previously called MachineInteger

47 matrix (John Abbott)

47.1 User documentation for the classes matrix, MatrixView and ConstMatrixView

CoCoALib offers two distinct concepts for dealing with matrices: one is an explicit implementation of a matrix, the other is a way to "view" an existing object as though it were a matrix (possibly of a special form). An example of a `MatrixView` is seeing a `std::vector<RingElem>` as a row matrix (see `MatrixViews` (Sec.51)).

There are two categories of matrix view, namely `ConstMatrixView` and `MatrixView`. The only difference between them is that the former does not allow you to change the entries while the latter allows you to change them (or at least some of them).

In contrast, a true `matrix` offers further operations for changing rows, columns and the dimensions – see the maintainer documentation if you're curious about why these operations are not allowed on a `MatrixView`.

Here are some guidelines for writing a function or procedure which takes matrices as arguments. If the function/procedure does not change the structure of the matrix, then use `ConstMatrixView` or `MatrixView`. If the structure of the matrix parameter may be modified then you must use `matrix&` as the parameter type.

47.1.1 Examples

- ex-matrix1.C
- ex-matrix2.C

47.1.2 Constructors and Pseudo-constructors

The following create a `matrix`:

- `NewDenseMat(R, r, c)` – (see `DenseMatrix` (Sec.18))
- `NewSparseMat(R, r, c)` – NOT YET IMPLEMENTED!!

The following create matrix *views* (e.g. changing an entry in `RowMat(v)` changes the vector `v`. See `MatrixViews` PseudoConstructors (Sec.51) for more details):

- `ZeroMat(R, r, c)`
- `IdentityMat(R, n)`
- `FilledMat(R, r, c, val)`
- `transpose(M)`
- `submat(M, rows, cols)`
- `ColMat(v)`
- `RowMat(v)`
- `DiagMat(v)`
- `BlockMat(A, B, C, D)`
- `ConcatVer(A, B)`
- `ConcatHor(A, B)`
- `ConcatDiag(A, B)`
- `ConcatAntiDiag(A, B)`

The following create a `matrix` and come from `MatrixSpecial` (Sec.50). See there for more details.

- `jacobian(f, indets)`
- `TensorMat(M1, M2)`

47.1.3 Operations on ConstMatrixView, MatrixView, matrix

- `BaseRing(M)` – the ring to which the matrix entries belong
- `NumRows(M)` – the number of rows in `M` (may be zero)
- `NumCols(M)` – the number of columns in `M` (may be zero)
- `out << M` – print the value of the matrix on ostream out (with a *dense* representation)
- `M1 == M2` – true iff `M1(i,j) == M2(i,j)` for all i,j
- `IsSymmetric(M)` – true iff `M(i,j) == M(j,i)` for all i,j
- `IsAntiSymmetric(M)` – true iff `M(i,j) == -M(j,i)` for all i,j
- `IsDiagonal(M)` – true iff `M(i,j) == 0` for all i!=j
- `IsMat0x0(M)` – true iff `NumRows(M) == 0 && NumCols(M)==0`

NB indices start from 0

- `M(i,j)` – the (`i`,`j`) entry of `M`
- `IsZeroRow(M,i)` – true iff row `i` of `M` is zero
- `IsZeroCol(M,j)` – true iff column `j` of `M` is zero

The following come from `MatrixArith` (Sec.48), see there for more details.

- `* + - /`
- `det(M)`
- `rank(M)`
- `inverse(M)`
- `adjoint(M)`
- `void mul(matrix& lhs, M1, M2)`
- `LinSolve(M,rhs)`
- `LinKer(M)`

47.1.4 Operations on MatrixView, matrix

- `M->myIsWritable(i,j)` – true iff posn `(i,j)` can be written to
- `SetEntry(M,i,j,val)` – set entry `(i,j)` of matrix `M` to `val` (integer, rational, RingElem). Throws `ERR::ConstMatEntry` if the entry is not writable
- `AssignZero(M)` – set all entries of `M` to zero
- `MV->myRawEntry(i,j)` – raw pointer to `(i,j)` entry (may be called only if the `(i,j)` posn is writable)
- `MV->myAssignZero()` – sets all entries to zero. Throws `ERR::ConstMatEntry` if not all entries can be made zero

NOTE: You cannot set a matrix entry the obvious way, *i.e.* `M(i,j) = value;` You must use `SetEntry(M,i,j,value)`. Calling `SetEntry` on a position which is not writable will throw `CoCoA::ERR::BadMatrixSetEntry`

47.1.5 Operations on matrix

With sanity checks

- `SwapRows(M,i1,i2)` – swap rows `i1` and `i2`
- `SwapCols(M,j1,j2)` – swap columns `j1` and `j2`
- `DeleteRow(M,i)` – delete row `i` and moves up the following rows
- `DeleteCol(M,j)` – delete column `j` and moves up the following cols

Without sanity checks

- `M->myResize(r,c)` – change size of `M` to `r`-by-`c` (new entries are zero)
- `M->myRowMul(i,r)` – multiply row `i` by `r`
- `M->myColMul(j,r)` – multiply column `j` by `r`
- `M->myAddRowMul(i1,i2,r)` – add `r` times row `i2` to row `i1`
- `M->myAddColMul(j1,j2,r)` – add `r` times column `j2` to column `j1`
- `M->mySwapRows(i1,i2)` – swap rows `i1` and `i2`
- `M->mySwapCols(j1,j2)` – swap columns `j1` and `j2`

NOTE: these are not permitted on `MatrixViews` (Sec.51) because of various problems which could arise *e.g.* with aliasing in block matrices (see maintainer documentation). `myResize` simply truncates rows/columns if they are too long, and any new entries are filled with zeroes. So, if you resize to a smaller matrix, you get just the "top left hand" part of the original.

At the moment assignment of matrices is not allowed. The only way to make a copy of a matrix (view) is by calling a genuine constructor (so far only `NewDenseMat` comes into this category).

47.1.6 Utility functions

- `IsRectangular(VV)` – says whether a `vector` of `vector` is rectangular

47.2 Library contributor documentation

The classes `ConstMatrixView` , `MatrixView` and `matrix` are just reference counting smart-pointers to objects of type derived from the abstract base classes `ConstMatrixViewBase`, `MatrixViewBase` and `MatrixBase` respectively; this is analogous to the way `ring` (Sec.69)s are implemented. Consequently every concrete matrix class or matrix view class must be derived from these abstract classes. At the moment, it is better to derive from `MatrixViewBase` rather than `ConstMatrixViewBase` because of the way `BlockMat` is implemented.

The base class `ConstMatrixViewBase` declares the following pure virtual member fns:

- `myBaseRing()` – returns the ring to which the matrix entries belong
- `myNumRows()` – returns the number of rows in the matrix
- `myNumCols()` – returns the number of columns in the matrix
- `myEntry(i,j)` – returns ConstRefRingElem aliasing the value of entry `(i,j)`
- `IamEqual(M)` – true iff *this==M
- `IamSymmetric()` – true iff entry (i,j) == entry (j,i)
- `IamAntiSymmetric()` – true iff entry (i,j) == -entry (j,i)
- `IamDiagonal()` – true iff entry (i,j) == 0 for i!=j
- `myMulByRow(v,w)` – v = w.M, vector-by-matrix product
- `myMulByCol(v,w)` – v = M.w, matrix-by-vector product
- `myIsZeroRow(i)` – true iff row `i` is zero
- `myIsZeroCol(j)` – true iff column `j` is zero
- `myDet(d)` – computes determinant into d
- `myRank()` – computes rank (matrix must be over an integral domain)
- `myOutput(out)` – print out the matrix on ostream `out`
- `myCheckRowIndex(i)` – throws an exception ERR::BadRowIndex if `i` is too large
- `myCheckColIndex(j)` – throws an exception ERR::BadColIndex if `j` is too large

These are the additional virtual functions present in `MatrixViewBase`:

- `myIsWritable(i,j)` – true iff entry `(i,j)` can be modified; `i` & `j` are unchecked
- `mySetEntry(i,j,value)` – set entry `(i,j)‘ to ‘‘value` (integer, rational, RingElem)
- `myAssignZero()` – set all entries to zero

These are the additional virtual functions present in `MatrixBase`:

- `myRowMul(i,r)` – multiply row i by r
- `myColMul(j,r)` – multiply column j by r
- `myAddRowMul(i1,i2,r)` –add r times row i2 to row i1
- `myAddColMul(j1,j2,r)` –add r times column j2 to column j1
- `mySwapRows(i1,i2)` – swap rows i1 and i2
- `mySwapCols(j1,j2)` – swap columns j1 and j2

Default definitions:

- IamEqual, IamSymmetric, IamAntiSymmetric, IamDiagonal, myMulByRow, myMulByCol, myIsZeroRow, myIsZeroCol, myOutput all have default *dense* definitions
- myDet and myRank have default definitions which use gaussian elimination

47.3 Maintainer documentation for the matrix classes

I shall assume that you have already read the User Documentation and Library Contributor Documentation.

The implementation underwent a big structural change in April 2008. I believe most of the design is sensible now, but further important changes could still occur. The implementation of the three matrix classes is wholly analogous to that of ring: they are simply reference counting smart-pointer classes (which may have derived classes). If assignment of matrices becomes permitted then some extra complication will be needed – e.g. `MakeUnique`, and the pointed object must be able to clone itself.

The only delicate part of the implementation is in `myMulByRow` and `myMulByCol` where a buffer is used for the answer so that the fns can be exception clean and not suffer from aliasing problems between the args.

Recall that by convention member functions of the base class do not perform sanity checks on their arguments; though it is wise to include such checks inside `CoCoA_ASSERT` calls to help during debugging. The sanity check should be conducted in the functions which present a nice user interface.

Q: Why did I create both `MatrixView` and `ConstMatrixView`?

A: Because the usual C++ *const mechanism* doesn't work the way I want it to. Consider a function which takes an argument of type `const MatrixView&`. One would not expect that function to be able to modify the entries of the matrix view supplied as argument. However, you can create a new non const `MatrixView` using the default copy ctor, and since `MatrixView` is a smart pointer the copy refers to the same underlying object. Currently, a `MatrixView` object does not perform *copy on write* if the reference count of the underlying object is greater than 1 – it is not at all clear what *copy on write* would mean for a matrix view (Should the underlying object be duplicated??? I don't like that idea!).

Q: Why are row, column and resizing operations which are allowed on `matrix` objects not allowed on `MatrixView` objects?

A: I disallowed them because there are cases where it is unclear what should happen. For example, suppose M is a true matrix, and someone creates the view MtM defined to be `ConcatHor(M, transpose(M))` then there is non-trivial aliasing between the entries of MtM. What should happen if you try to multiply the second row of MtM by 2? What should happen if you try to add a new column to MtM? In general, resizing MtM would be problematic. Here's another case: it is not clear how a resize operation should work on a matrix view based on a `vector<RingElem>`; would the underlying vector be resized too?

I chose to offer member fns for checking indices so that error messages could be uniform in appearance. I chose to have two index checking member fns `myCheckRowIndex` and `myCheckColIndex` rather than a single unified fn, as a single fn would have to have the *ugly* possibility of throwing either of two different exceptions.

I declared (and defined) explicitly the default ctor and dtor of the three base classes, to prohibit/discourage improper use of pointers to these classes.

The default *dense* definition of `MatrixBase::myOutput` seems a reasonable starting point – but see the bugs section below!

47.4 Bugs, Shortcomings and other ideas

The use of `std::vector<RingElem>` should be replaced by `ModuleElem` which automatically guarantees that all its components are in the same ring.

Should the default *dense* definitions of the output functions be removed? They could be quite inappropriate for a large sparse matrix.

Should the OpenMath output function send the ring with every value sent (given that the ring is also specified in the header)?

Should the index checking fns `myCheckRowIndex` and `myCheckColIndex` really throw? Perhaps there should be an alternative which merely returns a boolean value? When would the boolean version be genuinely beneficial?

Why can you not simply write `M(i,j) = NewValue;`? It is non-trivial because if M is a sparse matrix then use of `M(i,j)` in that context will require a structural modification to `M` if `NewValue` is non-zero and currently `M` has no `[i,j]` element. This natural syntax could be made possible by using a proxy class for `M(i,j)`; in a RHS context it simply produces a ConstRefRingElem for the value of the entry; in a LHS context the appropriate action depends on the implementation of the matrix.

I'm quite unsure about the signatures of several functions. I am not happy about requiring the user to use member functions for self-modifying operations (*e.g.* swap rows, etc) since elsewhere member functions by convention do not check the validity of their arguments.

Virtual member fn `myIsWritable` is not really intended for public use, but an arcane C++ rule prevents me

from declaring it to be `protected`. Apparently a `protected` name in the base class is accessible only through a ptr/ref to the derived class (and not through one to the base class) – no idea why!

Should assignment of matrices be allowed? Ref counting should make this relatively cheap, but must beware of the consequences for iterators (e.g. if it is possible to have a *reference to a row/column of a matrix*).

Would it be useful/helpful/interesting to have row-iterators and col-iterators for matrices?

47.5 Main changes

2012

- April: added SwapRows, SwapCols
- March: changed printing style

2011

- February: IsSymmetric, IsAntiSymmetric, IsDiagonal, operator== default and some optimized implementations.
- February (v0.9942): first release of `MatrixSpecial` (Sec.50) files

48 MatrixArith (John Abbott)

48.1 User documentation for MatrixArith

`MatrixArith` gathers together a number of operations on matrices; in most cases these operations are happy to accept a `MatrixView` (see `MatrixViews` (Sec.51)) as argument.

When not specified, a matrix argument is of type `ConstMatrixView`.

There are two ways of multiplying two matrices together. The infix operators return a `DenseMatrix` (Sec.18); the procedural version may be slightly faster than the infix operator.

- `mul(matrix& lhs, M1, M2)` – a procedure equivalent to `lhs = M1*M2;`, note that `lhs` might be a `SparseMatrix` **(not yet implemented)**
- `operator*(M1, M2)` – the product `M1*M2`
- `operator+(M1, M2)` – the sum `M1+M2`
- `operator-(M1, M2)` – the difference `M1-M2`
- `power(M, n)` compute `n`-th power of `M`; if `n` is negative then `M` must be invertible
- `operator*(n, M1)` – scalar multiple of `M1` by `n` (integer or RingElem)
- `operator*(M1, n)` – scalar multiple of `M1` by `n` (integer or RingElem)
- `operator/(M1, n)` – scalar multiple of `M1` by `1/n` (where `n` is integer or RingElem)
- `operator-(M1)` – scalar multiple of `M1` by -1

Here are some matrix norms. The result is an element of the ring containing the matrix elements. Note that `FrobeniusNorm2` gives the **square** of the Frobenius norm (so that the value surely lies in the same ring).

- `FrobeniusNorm2(M)` – the **square** of the Frobenius norm
- `OperatorNormInfinity(M)` – the infinity norm, ring must be ordered
- `OperatorNorm1(M)` – the one norm, ring must be ordered

Here are some fairly standard functions on matrices.

- `det(M)` – determinant of `M` (M must be square)

- `rank(M)` – rank of `M` (the base ring must be an integral domain)
- `inverse(M)` – inverse of `M` as a `DenseMatrix` (Sec.18)
- `adjoint(M)` – adjoint of `M` as a `DenseMatrix` (Sec.18)
- `PseudoInverse(M)` – PseudoInverse of `M` as a `DenseMatrix` (Sec.18). I suspect that it requires that the matrix be of full rank.
- `LinSolve(M,rhs)` – solve for `x` the linear system `M*x = rhs`; result is a `DenseMatrix` (Sec.18); if no soln exists, result is the 0-by-0 matrix
- `LinKer(M)` – solve for `x` the linear system `M*x = 0`; returns a `DenseMatrix` (Sec.18) whose columns are a base for `ker(M)`

Here are some standard operations where the method used is specified explicitly. It would usually be better to use the generic operations above, as those should automatically select the most appropriate method for the given matrix.

- `void det2x2(RingElem& d, M)` – for 2x2 matrices
- `void det3x3(RingElem& d, M)` – for 3x3 matrices
- `void DetByGauss(RingElem& d, M)`
- `RankByGauss(std::vector<long>& IndepRows, M)`
- `InverseByGauss(M)` – some restrictions (needs gcd)
- `AdjointByDetOfMinors(M)`
- `AdjointByInverse(M)` – base ring must be integral domain
- `LinSolveByGauss(M,rhs)` – solve a linear system using gaussian elimination (base ring must be a field), result is a `DenseMatrix` (Sec.18)
- `LinSolveByHNF(M,rhs)` – solve a linear system using Hermite NormalForm (base ring must be a PID), result is a `DenseMatrix` (Sec.18)
- `LinSolveByModuleRepr(M,rhs)` – solve a linear system using module element representation, result is a `DenseMatrix` (Sec.18)
- `void GrammSchmidtRows(MatrixView& M)` – NYI
- `void GrammSchmidtRows(MatrixView& M, long row)` – NYI

48.2 Maintainer documentation for MatrixArith

Most impls are quite straightforward.

`power` is slightly clever with its iterative impl of binary powering.

`LinSolveByGauss` is a little complicated because it tries to handle all cases (*e.g.* full rank or not, square or more rows than cols or more cols than rows)

48.3 Bugs, Shortcomings and other ideas

Can we make a common "gaussian elimination" impl which is caled by the various algorithms needing it, rather than having several separate implementations?

Is the procedure `mul` really any faster than the infix operator?

48.4 Main changes

2012

- June: Added negation, multiplication and division of a matrix by a scalar.
- April: Added LinSolve family (incl. LinSolveByGauss, LinSolveByHNF, LinSolveByModuleRepr)

2011

- May: Added power fn for matrices: cannot yet handle negative powers.
- March: added multiplication by RingElem

49 MatrixForOrdering (Anna Bigatti)

49.1 User Documentation

This is very preliminary documentation. These functions are about matrices which define term orderings. They expect and return matrices over `RingZZ` (Sec.79).

49.1.1 Examples

- ex-OrderingGrading1.C

49.1.2 PseudoConstructors

- `NewPositiveMat(M)` – returns a matrix with non-negative entries which defines an equivalent term-ordering (but grading may be different!)
- `NewMatMinimize(M)` – the (ordering) matrix obtained by removing linearly dependent rows
- `NewDenseMatRevLex(n)` – produce the `n`-by-`n` dense matrix over `RingZZ` (Sec.79) corresponding to the revlex ordering on `n` indets
- `NewMatCompleteOrd(ConstMatrixView M)` – complete M to an ordering matrix; if M is suitable the resulting matrix defines a term-ordering.
- `NewMatCompleteOrd(ConstMatrixView M, ConstMatrixView NewRows)` – concatenate M and NewRows, and remove redundant rows.
- `NewMatElim(NumIndets, IndetsToElim)` – returns the dense matrix for the elimination ordering of the given indets
- `NewMatElim(GradingM, IndetsToElim, IsHomog)` – ???
- `NewHomogElimMat(GradingM, IndetsToElim)` – ???

49.1.3 Queries

- `IsTermOrdering(M)` – true iff matrix `M` defines a term ordering
- `IsPositiveGrading(M)` – true iff `M` defines a positive grading (i.e. no null columns and first non-zero entry in each column is positive)
- `IsPositiveGrading(M, GradingDim)` – true iff the first `GradingDim` rows of `M` define a positive grading

49.2 Maintainer Documentation

49.3 Bugs, Shortcomings, and other ideas

Doc is woefully incomplete.

Definitely don't like the name NewMatMinimize!

Fixed 2009-09-22: Naming convention not respected ("Matrix" should be "Mat")

50 MatrixSpecial (Anna Bigatti)

50.1 User documentation for MatrixSpecial

50.1.1 Examples

50.1.2 Special Matrices

Jacobian Matrix

The (i,j)-th element of the Jacobian matrix is defined as the derivative of i-th function with respect to the j-th indeterminate.

- `jacobian(f, indets)` – where `f` (polynomials) and `indets` (indeterminates) are vectors of `RingElem` (Sec.71), all belonging to the same `PolyRing` (Sec.63). Throws if both `f` and `indets` are empty (cannot determine the `ring` (Sec.69) for constructing the 0x0 `matrix` (Sec.47)).

Tensor Product of matrices

a_11 B	a_12 B	...	a_1c B
a_21 B	a_22 B	...	a_2c B
		...	
a_r1 B	a_r2 B	...	a_rc B

- `TensorMat(A, B)` – where `A` and `B` are matrices with the same BaseRing.

50.2 Maintainer documentation

50.3 Bugs, shortcomings and other ideas

50.4 Main changes

2011

- February (v0.9942): first release (`jacobian`)
- March (v0.9943): added `TensorMat`

51 MatrixViews (John Abbott)

51.1 User documentation for MatrixViews

A `MatrixView` offers a means to **view** one or more **existing objects** as though they were a `matrix` (Sec.47):

- if you change the entries in the objects then the `MatrixView` changes;
- if you change the entries in the `MatrixView` then the objects change;
- if you destroy or change the structure of the objects then the `MatrixView` may become invalid (and using it could lead to the dreaded *undefined behaviour*, i.e. probably a crash).

51.1.1 Examples

- ex-matrix1.C
- ex-matrix2.C

NB The *views* do not make copies, so be careful with temporaries! Look at these examples (`val` is a `RingElem` (Sec.71)):

```
// OK
const vector<RingElem> v(3, val);
MatrixView MV = RowMat(v);  // MV reads/writes in the vector v

// NO   this compiles, but the vector disappears after the ";"!!
ConstMatrixView MVGhost = RowMat(vector<RingElem>(3, val));

// OK   NewDenseMat makes a copy of the vector before it disappears
matrix M = NewDenseMat(RowMat(vector<RingElem>(3, val)));
```

51.1.2 Pseudo-constructors

Constant MatrixViews

NB no entry is *writable*

- `ZeroMat(R, r, c)` – constant `r`-by-`c` zero matrix over `R`
- `IdentityMat(R, n)` – constant `n`-by-`n` identity matrix over `R`
- `FilledMat(R, r, c, val)` – constant `r`-by-`c` matrix over `R` filled with val (a `RingElem` (Sec.71), long, `BigInt` (Sec.8), or `BigRat` (Sec.9))

MatrixViews of a vector

You can *view* a `std::vector<RingElem>`, all of whose entries belong to the same `ring` (Sec.69), as a matrix in three ways:

- `ColMat(v)` – view a `vector<RingElem> v` as a column matrix
- `RowMat(v)` – view a `vector<RingElem> v` as a row matrix
- `DiagMat(v)` – view a `vector<RingElem> v` as a diagonal matrix (NB: only the diagonal entries are *writable*)

MatrixViews of a matrix

- `transpose(M)` – transposed *view* of the matrix `M`
- `submat(M, rows, cols)`– submatrix *view* into `M`; the rows and columns visible in the submatrix are those specified in the arguments `rows` and `cols` (which are of type `std::vector`)

MatrixViews of more matrices

The following pseudo-constructors assemble several matrices into a bigger one; the argument matrices must all have the same `BaseRing`.

- `ConcatVer(A, B)` – matrix *view* with the rows of `A` above those of `B`

A
B

- `ConcatHor(A, B)` – matrix *view* with the cols of `A` before those of `B`

A	B

- `ConcatDiag(A,B)` – block diagonal matrix *view*

A	0
0	B

- `ConcatAntiDiag(A,B)` – block antidiagonal matrix *view*

0	A
B	0

- `BlockMat(A, B, C, D)` – block matrix *view*

A	B
C	D

NB the boundaries of the four submatrices must be aligned.

51.1.3 Operations on ConstMatrixView, MatrixView

See `matrix` operations (Sec.47)

51.2 Maintainer documentation for MatrixViews

Most of the implementations are quite straightforward; the tricky part was getting the design of the abstract classes right (well, I hope it is right now). Below are a few comments on some less obvious aspects of the implementations.

Note: it is a mathematical fact that the determinant of the 0x0 matrix is 1.

`ZeroMatImpl` and `IdentityMatImpl` are both derived from `MatrixViewBase` rather than `ConstMatrixViewBase` as one might naturally expect. The main reason for this is to simplify the implementation of `BlockMat` views. I wanted to be lazy and implement `ConcatDiag` and `ConcatAntidiag` using `BlockMat`; while this may not be the best implementation, it is a natural approach and should certainly work as one might reasonably expect. However, the pseudo-ctor `BlockMat` has just two signatures: if any one of the submatrices is *const* then whole result becomes *const*. I didn't want to implement sixteen different signatures for `BlockMat`, and the easy way out seemed to be to make `ZeroMatImpl` and `IdentityMatImpl` non-const. As a consequence there are a number of *useless* member functions in `ZeroMatImpl` and `IdentityMatImpl`. I believe this compromise is reasonable. It seemed reasonable to allow `ZeroMatImpl::myAssignZero` to succeed.

There is a small problem with creating a matrix from an empty `std::vector` because there is no indication of what the base ring should be. I have chosen to throw an error if one tries to create a matrix view from an empty vector (in `RowMat`, `ColMat` and `DiagMat`).

The routines which access the (i,j) entry in a BlockMat are messy. I could not see an elegant way to make them simpler (or to avoid repeating similar structure in several places in the code). See Bugs about implementing `BlockMat` in terms of `ConcatVer` and `ConcatHor`.

51.3 Bugs, Shortcomings and other ideas

There is an appalling amount of code duplication in the implementations. I do not yet see a good way of reducing this. I hope someone will sooner or later find an elegant way to avoid the duplication. Maybe a *diagonal* abstract class for ZeroMatImpl, IdentityMatImpl, DiagMatImpl, ConstDiagMatImpl?

It is a great nuisance to have to implement two very similar classes: one for the *const* case, and the other for the *non-const* case. Is there a better way?

Add `ColMat`, `RowMat` and `DiagMat` for a free module element?

Should `submatrix` allow repeated row/col indices? It could lead to some some funny behaviour (e.g. setting one entry may change other entries), so perhaps it would be better to forbid it? Currently, it is forbidden.

The pseudo-ctor for `submatrix` ought to accept begin/end iterators instead of insisting that the caller put the indices in `std::vectors`.

Should there be a more general version of `BlockMat` which allows non-aligned borders? `BlockMat` could be eliminated and replaced by suitable calls to `ConcatVer` and `ConcatHor`.

Tensor product of two matrices: we implement it as a DenseMatrix instead of MatrixView because the latter would give no practical advantage and hide the cost of accessing the entries.

51.4 Main changes

2011

- February (v0.9943):
  - optimized implementations for `IsSymmetric`, `IsAntiSymmetric`, `IsDiagonal`, `operator==`
  - added `FilledMat` -

52 MemPool (John Abbott)

52.1 User Documentation for MemPool

52.1.1 General description

A MemPool provides a simple and fast memory management scheme for memory blocks of **fixed size**. It is particularly well-suited to cases where there are many interleaved allocations and deallocations. You probably do not need to know about MemPool unless you plan to write some *low-level* code.

`MemPool`s work by acquiring large *loaves* of memory from the system, and dividing these loaves into *slices* of the chosen size. A simple free-list of available slices is maintained. New loaves are acquired whenever there are no slices available to meet a request. Note that the space occupied by the loaves is returned to the system only when the `MemPool` object is destroyed. Also note that a `MemPool` simply forwards to `::operator new` any request for a block of memory of size different from that specified at the creation of the `MemPool` object; wrong size deallocations are similarly forwarded to `::operator delete`.

52.1.2 Basic Use

The constructor for a `MemPool` requires that the size (in bytes) of the blocks it is to manage be specified (as the first argument). We recommend that the `MemPool` be given a name (second argument as a string); the name is useful only for debugging. The third argument may occasionally be useful for more advanced use.

```
MemPool workspace(16); // 16 byte slices used as temporary workspaces

MemPool MemMgr(sizeof(widget), "memmgr for widgets");
```

Once the `MemPool` has been created, a new block of memory is obtained via a call to the member function `alloc`, and a block of memory is freed via a call to the member function `free` (only to be applied to blocks previously allocated by the same `MemPool`). In fact, `alloc` and `free` have two variants:

```
MemPool::alloc()  allocates a block of the default size for the ``MemPool''

MemPool::alloc(sz) allocates a block of ``sz'' bytes; if ``sz'' is not the
default size for the ``MemPool'' the request is passed on to ``::operator new''

MemPool::free(ptr) frees a default sized block with address ``ptr''

MemPool::free(ptr, sz) frees a block of ``sz'' bytes with address ptr, if
``sz'' is not the default size for the ``MemPool'' the request is passed on to
``::operator delete''
```

The variants taking an explicit block size permit `MemPool`s to be used by a class specific operator new/delete pair (see example program below). In particular, it is not an error to ask a `MemPool` for a block of memory whose size differs from the size declared when the `MemPool` was constructed; indeed, this is a necessary capability if the `MemPool` is to be used inside operator new/delete. Attempting to `alloc` too much memory will result in a `std::bad_alloc` exception being thrown.

If you encounter bugs which may be due to incorrect memory management then `MemPool` has some facilities to help you detect various common bugs, and isolate their true causes. How to do this is described in the following section *Debugging with MemPools*.

It is possible to get some crude logging information from a `MemPool`. The global variable `MemPoolFast::ourInitialVerbosityL` indicates the verbosity level for newly created `MemPool`s; the verbosity level of individual `MemPool` objects may be set explicitly by calling the member function `SetVerbosityLevel`. The various verbosty levels are described below in the section entitled *The Verbosity Levels*.

Technical note: `MemPool` is just a typedef for the true class name `MemPoolFast` (or `MemPoolDebug` if you enable debugging).

52.1.3 Debugging with MemPools

The preprocessor variable `CoCoA_MEMPOOL_DEBUG` can be set at compile-time to perform run-time checks and obtain debugging information and statistics. Note that **recompilation of all source files** depending on `MemPool` will be necessary. When the preprocessor variable is set the typedef `MemPool` refers to the class `MemPoolDebug` – throughout this section we shall speak simply of `MemPool`.

Each `MemPool` object maintains a record of its own level of verbosity and debug checks. Upon creation of a new `MemPool` object these levels are set automatically to the values of these two global variables:

```
MemPoolDebug::ourInitialDebugLevel
MemPoolDebug::ourInitialVerbosityLevel
```

The values of these globals should be set **before creating** any MemPools, *i.e.* before creating the `GlobalManager` (which creates the MemPools for the ring of integers and the rationals).

The ostream on which logging data is printed defaults to `std::clog` but may be changed to another ostream via a call like `MemPoolSetLogStream(LogFile)`; the logging stream is global, *i.e.* the same for all MemPools.

Similarly the ostream on which error logs are printed defaults to `std::cerr` but may be changed to another ostream via a call like `MemPoolSetErrStream(ErrFile)`; the error stream is global, *i.e.* the same for all MemPools.

After construction of a `MemPool` object its levels can be adjusted using the member functions:

```
MemPool MemMgr(...);         // construct MemPool
MemMgr.SetDebugLevel(n);     // change debug level for this object
MemMgr.SetVerbosityLevel(n);// change verbosity level for this object
```

You can arrange for a `MemPool` to print out some summary statistics at regular intervals. The interval (in seconds) used for such messages is approximately the value of

```
MemPoolDebug::ourOutputStatusInterval
```

52.1.4 The Verbosity Levels

To help in debugging and fine tuning, you can get some logging messages out of a `MemPool`; these are printed on `GlobalLogput` (see `io` (Sec.42)). Here is a description of the various levels of *verbosity*:

Level 0 No logging information is produced (but error messages may be produced if debugging is active, see below)

Level 1 A brief message is produced upon creation of each `MemPool` object; and another upon destruction (including some summary statistics).

Level 2 In addition to level 1: a log message is produced for each new loaf allocated by a `MemPool`, including some summary statistics. This may be useful to monitor how much memory is being allocated, and how quickly.

Level 3+ In addition to level 2: a log message is produced for each allocation and deallocation of a block by a `MemPool`; this can be used to isolate memory leaks (see comment below).

52.1.5 Using Verbosity Level 3

This is a very verbose level: each allocation/deallocation gives rise to a printed message (on a single rather long line). These messages can be analyzed to help isolate when a leaked block of memory is allocated; or, in conjunction with debug level 1, it can help find when a block of memory which is written to after being freed was allocated. Note that this can produce enormous amounts of output, so you are advised to send logging output to a file. The output may be processed by the program `leak_checker` (in this directory) to help track down memory leaks: see the user documentation in leak_checker.txt

Each message about an alloc/free contains a sequence number: there are separate counts for calls to `alloc` and calls to `free`. If the leak_checker program indicates that there is no matching `free` for the N-th call to `alloc` then the N-th call to `alloc` for that particular `MemPoolDebug` object can be intercepted easily in a debugger by setting a breakpoint in the function `MemPoolDebug::intercepted`, and by calling the member function `InterceptAlloc` with argument N at some point before the N-th call to `alloc`. The N-th call to `free` can be intercepted in an analogous way by calling instead the member function `InterceptFree`. It is probably a good idea to call `InterceptAlloc` or `InterceptFree` as soon as you can after the `MemPoolDebug` object has been created; of course, recompilation will be necessary.

52.1.6 Debug Levels in MemPools

If CoCoA_MEMPOOL_DEBUG was set during compilation then each `MemPool` object performs some debug checking. If the checks reveal a problem then an error message is printed on `GlobalErrput`. Upon creation of a `MemPool` object, the debug level is set to the value of the global variable:

```
MemPoolDebug::ourInitialDebugLevel
```

After creation the debug level can be adjusted by calling the member function `SetDebugLevel`; this must be called before the `MemPool` has allocated any space. Any attempts to change the debug level are silently ignored after the first allocation has been made.

Here are the meanings of the different levels of checking: (each higher level includes all lower levels)

Level 0 A count of the number of allocations, deallocations and *active* blocks is maintained: a block is *active* if it has been allocated but not subsequently freed. The only check is that the number of active blocks is zero when the `MemPool` object is destroyed; an error message is printed out only if there are some active blocks. This level is rather faster than the higher levels of debugging, but should detect the existence of leaked memory; higher levels of debugging will probably be necessary to isolate the cause of any leak.

Level 1 This level should detect several types of common error: writing just outside the allocated region, writing to a block shortly after freeing it, perhaps reading from a block shortly after freeing it, trying to free a block not allocated by the given `MemPool` object, perhaps reading from an uninitialized part of an allocated block. Freeing a zero pointer via a `MemPool` is also regarded as worthy of a warning.

When a block of memory is allocated it is filled with certain values (including small *margins* right before and after the requested block). The values in the margins are checked when the block is freed: anything unexpected produces an error message. A freed block is immediately filled with certain other values to help detect reading/writing to the block after it has been freed. These values are checked when the block is next reallocated.

Level 2 This level has not been tested much. It will probably be very much slower than any lower level, and is intended to help track down cases where a freed block is written to some time after it has been freed. A freed block is never reallocated, and all freed blocks are checked for being written to each time `alloc` or `free` is called; an error message is printed if a modified freed block is found. You need to be pretty desperate to use this level. A corrupted freed block is cleared to its expected *free* state as soon as it is reported – so persistent writing to a freed block can be detected.

52.1.7 Example: Using a MemPool as the memory manager for a class

Suppose you already have a class called MyClass. Here are the changes to make so that heap-located instances of MyClass reside in slices managed by a MemPool; obviously stack-located instances cannot be managed by MemPool.

Add in the definition of MyClass (typically in the file MyClass.H):

```
private:
 static MemPool myMemMgr;

public:
 static inline void operator delete(void* DeadObject, size_t sz)
 { myMemMgr.free(DeadObject, sz); }
 inline void* operator new(size_t sz)
 { return myMemMgr.alloc(sz); }
```

The class static variable must be defined in some `.C` file, probably MyClass.C is the most suitable choice:

```
MemPool MyClass::myMemMgr = MemPool(sizeof(MyClass));
  or
MemPool MyClass::myMemMgr = MemPool(sizeof(MyClass), PoolName);
  or
MemPool MyClass::myMemMgr = MemPool(sizeof(MyClass), PoolName, NrWordsInMargin);
```

`PoolName` is a string: it is used only in logging and error messages in debugging mode, but it might be useful when debugging even when CoCoA_MEMPOOL_DEBUG is not defined; the default name is `Unnamed-MemPool`.

`NrWordsInMargin` is used only with debugging, and can be used to alter the width of the buffer zones placed before and after each slice (default=4).

Here is a simple example program showing how `MemPool`s can be used, and how the debugging facilities can be employed. Compile this program with `CoCoA_MEMPOOL_DEBUG` set, and then run it to see the error messages produced indicating improper use of memory resources.

```
#include <cstddef>
#include <iostream>
#include <string>
#include "CoCoA/MemPool.H"

using CoCoA::MemPool;
using namespace std;

class Date
{
 public:
  static void operator delete(void* DeadObject, size_t sz);
  void* operator new(size_t sz);

  Date(int d=1, int m=1, int y=1900, char app[40]="??");
  ~Date() {};
  Date& operator=(const Date& rhs);
  friend ostream& operator << (ostream& cout, const Date& D);

 private:
  static MemPool date_mempool;
  int day, month, year;
  char appointment[40];
};

// Define new versions of new and delete for Date...
inline void Date::operator delete(void* DeadObject, size_t sz)
{
  date_mempool.free(DeadObject, sz);
}

inline void* Date::operator new(size_t sz)
{
  return date_mempool.alloc(sz);
}

// We must initialize the static member Date::date_mempool...
MemPool Date::date_mempool = MemPool(sizeof(Date), "Date_Pool", 4);

//------------------------------------------------------------------------//
Date::Date(int d, int m, int y, char app[40])
{
  day = d;
  month = m;
  year = y;
  strcpy(appointment, app);
}

//------------------------------------------------------------------------//

Date& Date::operator=(const Date& RHS)
{
  if (this == &RHS) return *this;
```

```
  day = RHS.day;
  month = RHS.month;
  year = RHS.year;
  strcpy(appointment, RHS.appointment);
  return *this;
}

ostream& operator << (ostream& cout, const Date& D)
{
  cout << D.day << " " << D.month << ", " << D.year << " \t";
  cout << "appointment: " << D.appointment;

  return cout;
}

//------------------------------  main  ------------------------------//

int main()
{
  cout << endl << "== EXAMPLE ==" << endl << endl;
  const int N = 4000;

  Date *D1[N], *D2, *D3;

  D2 = new Date;
  (*D2) = Date(6,12,1965, "compleanno");  cout << "*D2 = " << *D2 << endl;
  D3 = new Date;  cout << "*D3 = " << *D3 << endl;

  delete D2;
  delete D2;  // ERROR!  D2 already freed

  for ( int i=0 ; i<N ; i++ )    D1[i] = new Date;
  for ( int i=N-1 ; i>=0 ; i-- )  delete  D1[i];

  Date *D8 = new Date[4];
  D8[0] = Date(1,4,2001, "pesce d'Aprile");
  delete D8;  // ERROR!  D8 not allocated by mempool
  // D3 not deleted -- will be detected when mempool is destroyed
  return 0;
}
```

52.2 Maintenance notes for the MemPool source code

The code for `MemPoolFast` and `MemPoolDebug` is exception-safe. The only exception this code could cause is `std::bad_alloc` in the member functions `MakeNewLoaf` or by a forwarded call to `::operator new` inside the member functions `alloc`.

The class `MemPoolFake` simply forwards all allocation/deallocation calls to `::operator new/delete`. It was added hastily to enable a threadsafe compilation (assuming that `::operator new` and `::operator delete` are themselves threadsafe).

The idea of `MemPool`s was taken from *Effective C++* by Scott Meyers, but the code here has evolved considerably from what was described in the book.

There are two virtually independent implementations: one for normal use, and one for use while debugging, the selection between the two versions is determined by the preprocessor symbol CoCoA_MEMPOOL_DEBUG: if this symbol is undefined then `MemPool` is a typedef for `MemPoolFast` otherwise it is a typedef for `MemPoolDebug`.

`MemPoolDebug` uses internally a `MemPoolFast` object to handle the genuine memory management operations while `MemPoolDebug` performs validity checks and maintains counters for various sorts of operation.

52.2.1 MemPoolFast and loaf

The most important member functions of `MemPoolFast` are `alloc` and `free` for slices of the requested size; it is vital that these be fast (on average). Amazingly, no worthwhile gain in speed was observed when I made these functions inline; sometimes inline was noticeably slower (g++ oddity?). Anyway, for simplicity I have kept them out-of-line.

The idea behind a `MemPoolFast` is quite simple: unused slices are strung together in a *free list*, the last unused slice contains a null pointer. So `alloc` simply returns a pointer to the first slice in the free list, while `free` inserts a new slice at the front of the free list. The ctor makes sure that each slice is big enough to hold at least a pointer; the first part of a free slice is used to hold the pointer to the next free slice (any remaining space in a free slice is unused).

Note that there is a conundrum in choosing the right C++ type for the slices of a loaf, since the values kept in unused slices are pointers to slices, and there is no C++ type which is a pointer to itself. The type chosen for these entries is `void**`: this conveys the information that they are not pointers to C++ objects while also allowing pointer arithmetic (which is not allowed on values of type `void*`). Nonetheless the code is necessarily peppered with casts (to convert a `void***` into a `void**`); these are necessarily `reinterpret_cast`s but should be absolutely safe since they are only ever applied to genuine pointers to values (or to the null pointer). Actually the `reinterpret_cast`s could probably be replaced by two nested `static_cast`ss passing via the type `void*` but this would not help readability in the slightest.

What happens when a new slice is requested when the free list is empty? A new `loaf` is created, and cut into slices which are linked together to form a free list. A `loaf` is little more than a large chunk of raw memory acquired from the system (see below for more details). Note that if several loaves are in use then the freed slices from different loaves are strung together in a single free list; no attempt is made to keep slices from different loaves separate. In particular, no check is made for a loaf all of whose slices are unused; loaves are returned to the system only when the `MemPool` is destroyed.

Most of the data members of `MemPoolFast` are simple and with an obvious role. Here are a few observations about aspects which may not be completely obvious.

The data member `myLoaves` is an `auto_ptr` so that the class dtor can be simple; it also expresses the idea that the loaves pointed to are owned by the `MemPoolFast` object. Note that each `loaf` has a next pointer which is also an `auto_ptr`, so destroying the first loaf will destroy them all. I could not use a `std::list` because `loaf` does not have a copy ctor.

The data member `myFillNewLoaf` is used only when a new loaf is created (in `MakeNewLoaf`). If the flag is set, the slices in a new loaf are filled with the sentinel value expected by `MemPoolDebug`, i.e. `MEMPOOL_FREE_WORD`. This seemed the least obnoxious way of achieving the necessary behaviour.

The data member `myVerbosityLevel` was added to allow some minimal logging of resource consumption even with `MemPoolFast` objects: a brief message is output whenever a new loaf is acquired. It does complicate the class rather, but may be useful sometimes.

The only member functions exhibiting some complexity are: `myOutputStatus` uses a loop to count how many freed slices there are in each loaf, and the print out the results in `GlobalLogput`.

`MakeNewLoaf` first decides roughly how many slices the new loaf should have; creates the loaf, and inserts at the front of the list of loaves; prints out a logging message if required.

The separation of the class `loaf` from the class `MemPoolFast` is partly a historical accident – a side-effect of the tortuous search for a tolerably clean implementation. Overall, I regard it as a fairly happy accident because no details of the the class `loaf` are visible in the header file.

The class `loaf` has a simple primary role: it owns the raw memory acquired from the system. Destroying a `loaf` returns the raw memory to the system. Unfortunately the implementation became rather complicated. Each `loaf` contains a *next pointer* so that `loaf`s can be linked together in a list. I could not use a `std::list` since a `loaf` does not have a copy ctor (nor assignment); I prefer not to play dangerous games with copy ctors which destroy their arguments (non-standard semantics), and a clean copy ctor would probably be horribly inefficient. The *next pointer* is an `auto_ptr` so that destroying the first `loaf` in a list will actually destroy all of the `loaf`s in that list.

To fulfil a request for logging information about utilization of slices in each `loaf`, I added four member functions:

```
IamOriginator      - true iff arg points to a slice of this loaf
myFreeCounterReset - reset counters to zero in this loaf list
myCountFreeSlice   - incr my counter if slice is mine, o/w pass to next loaf
myOutputStatus     - print out utilization stats.
```

Apart from `IamOriginator`, I would much rather these functions did not exist.

The implementation of a `loaf` is straightforward (but a bit messy).

52.2.2 MemPoolDebug

The idea behind `MemPoolDebug` is that it should offer the same interface as `MemPoolFast` but will additionally perform validity checks and accumulate utilization statistics and print logging messages (if requested). The implementation is quite straightforward but rather long and messy as the code offers several levels of debug checks and logging message verbosity.

The idea behind a `MemPoolDebug` is that it manages slices in a manner which should help uncover incorrect use of memory: a newly allocated slice is filled with peculiar values (in case you read without first writing a sensible value there), a freed slice is immediately filled with an other peculiar value (in case you read after freeing), each slice has a small protective margin right before and after it (in case you write just outside the valid address range)... (the fill values are intended to be invalid as pointers, to help detect pointer following in *uninitialized memory*)

A count is kept of the number of `alloc` and `free` calls. This can help discover that some value was never freed, or maybe was freed twice. These counts are of type `size_t`, so they could overflow; but then you'd be a bit daft to try to debug such a large example, wouldn't you?

The default initial debugging and verbosity levels can be modified by setting the values of certain global variables – these value are respected only if you compiled with `CoCoA_MEMPOOL_DEBUG` set or if you used explicitly the class `MemPoolDebug` rather than the typedef `MemPool`. These values are consulted only when a `MemPoolDebug` object is created. Using global variables like this make its easy to vary the debug level (without having to recompile the whole library).

- `MemPoolDebug::ourInitialVerbosityLevel` default verbosity level
- `MemPoolDebug::ourInitialDebugLevel` default debug level
- `MemPoolDebug::ourDefaultMarginSize` default margin size (see below)
- `MemPoolDebug::ourOutputStatusInterval` print utilization statistics at roughly this interval (in seconds)

All the genuine memory management operations are handled by `myMemMgr`, a `MemPoolFast` object belonging to the `MemPoolDebug` object. This approach avoids having two similar copies of rather delicate code.

The margin size must be fixed in the ctor because `myMemMgr` needs to know what size slices it must manage. The margin size for a `MemPoolDebug` object cannot be changed later. Distinct `MemPoolDebug` objects may have different margin sizes.

The debug level may be changed after construction provided no slices have been issued; trying to make the various debug levels compatible would require very careful checking (which I cannot be bothered to do).

The verbosity level can be changed at any time (since there is no reason not to allow this).

The data member `myAliveOrDead` was added to help protect against attempts to use an already deleted `MemPoolDebug` object. All public member functions check that the field `myAliveOrDead` contains the expected value before proceeding: a CoCoALib error is thrown if the value is wrong. The correct value for a live `MemPoolDebug` object is the constant `MemPoolDebug::AliveMark`.

The data member `myHeadOfUsedList` is used at the highest level of debugging. All freed slices are placed on this list so they cannot be reissued to the user. Every call then scans all these freed slices to make sure they contain the correct fill value. This is intended to help discover writes to freed memory long after the slice has been freed. This level gets very slow on larger examples.

52.3 Bugs, Shortcomings, etc

Idea for better locality of reference: keep two free lists, one for the most recent loaf, and one for all older loaves. When most recent loaf fills up, create and use a new loaf unless the free list for all the older loaves exceeds 0.5 times the size of the most recent loaf. Not sure what to do if the freelist for old loaves is very long.

Add a new member function which *tidies up* the list of freed blocks? This might lead to better locality of reference, and ultimately to better run-time performance if called judiciously.

Could it be worth trying to help preserve locality of reference? Maybe freed slices could be returned to their own loaves. Properly nested alloc/free calls ought to preserve locality anyway.

Perhaps the globals `ourInitialDebugLevel` and `ourInitialVerbosityLevel` could be set inside the ctor for `GlobalManager`??

Member functions of `MemPoolFast/Debug` do not have names in accordance with the coding conventions. Cannot decide when I should use `void*` and when I should use `slice_t` for the arg types.

A potentially useful function could be one which tells the MemPool to check that it is empty (i.e. all allocated blocks have been freed). This is currently implicit in the debugging-mode dtor.

It might be an idea to maintain a registry of all existing MemPools, so that they can be told towards the end of the run that they should all be empty. Otherwise any MemPool which is never destroyed can never give an indication of any leaks of its own slices.

Could there be alignment problems with funny margin sizes? What about machines where pointers are a different size from `int`s?

The code may silently increase the size of requested blocks so that their lengths are integer multiples of the size of a `slice_t`. This does mean that writes outside the requested block but within the silently extended block are not detected (in debugging mode) – I guess that most block sizes are exact multiples anyway, so there is unlikely to be any problem in most practical situations.

Is the function `AlreadyFreed` working as one would expect? Currently it checks that the margins are those of a freed block, and uses that as the determining criterion. The argument is that an attempt to free a block suggests that user probably thought it hadn't been freed and so the user accessible data area is quite probably corrupted (i.e. not simply full of `MEMPOOL_FREE_WORD` values). I have also added a call to `OverwriteFreeCheck`, so that freeing an overwritten already freed block will cause two error messages to be printed. Previously, `AlreadyFreed` required that the data area be in tact for the block to count as already having been freed; an overwritten freed block would then be detected as an allocated block with corrupted margins. Maybe a memory map for an overwritten freed block would be a useful addition? (similar to that produced for an allocated block with corrupt margins).

The periodical printing of stats is rather crude. To make it more sophisticated will just made the code even more complex though (sigh).

`AutoPtrSlice` is still very experimental.

53 module (John Abbott)

53.1 User documentation for the classes module, ModuleBase, ModuleElem

You may also wish to look at the documentation for `FGModule` (Sec.30) the type which represents (explicitly) Finitely Generated Modules.

The classes `module`, `ModuleBase` and `ModuleElem` are closely linked together (analogously to the triple `ring`, `RingBase` and `RingElem`).

The class `module` is a reference counting smart pointer to an object of type derived from `ModuleBase`; all concrete types for representing modules are derived from `ModuleBase`. For a library implementor the class `ModuleBase` defines the minimal interface which every concrete module class must offer; indeed the concrete class must be derived from `ModuleBase`.

A user of CoCoALib who does not wish to add to the library need know only what it is in this section.

Analogously to `ring`s and `RingElem`s, every `ModuleElem` belongs to some `module`. So before you can compute with `ModuleElem`s you must create the `module`(s) which contain them.

To create a `module` you must a pseudo-constructor for one of the concrete module classes (refer to their documentation for details): *e.g.*

```
NewFreeModule(R, n)   -- create a new FreeModule of n components over R
```

The functions which one may apply directly to a module are:

```
NumCompts(M) -- the number of components an element of M has
BaseRing(M)  -- the base ring of M (i.e. M is a module over this ring)
gens(M)      -- a read only C++ vector containing the generators of M
zero(M)      -- a read only ModuleElem which is the zero of M
M1 == M2     -- are the two modules identical (same repr in memory)?
M1 != M2     -- opposite of M1 == M2
```

As you can see there is not a lot one can do to a module. Primarily they exist to "give the correct type" to module elements; internally they play a crucial role in applying operations to module elements. A C++ value of type `ModuleElem` represents an element of some concrete module. The module to which the value belongs is called the `owner` of that value. The owner of an object of type `ModuleElem` must be specified (explicitly or implicitly) when it is created, and cannot be changed during the lifetime of the object; the value it contains may, however, be changed (C++ const rules permitting).

Functions on ModuleElems

Let `v` be a non-const `ModuleElem`, and `v1, v2` be `const ModuleElems` all belonging to the same concrete module `M`. Let `R` be the base ring of `M`, and `r` a const element of `R`. Then we summarize the possible operations using C++ syntax:

```
owner(v1)   // gives the module to which v1 belongs

-v1                                     // Usual arithmetic operations
v1 + v2                  v1 - v2        // between ModuleElems and
r * v1                   v1 * r         // RingElems.
                         v1 / r
v = v1
v += v1                  v -= v1
v *= r                   v /= r
v1 == v2                 v1 != v2
IsZero(v1)               cout << v1

v[pos]  // throws if the module is not FGModule
```

In every case it is an error to combine/compare `ModuleElems` belonging to different modules. As you would expect, instead of multiplying or dividing by a `RingElem` (Sec.71) you may also multiply or divide by a machine integer, a `BigInt` (Sec.8) or a `BigRat` (Sec.9).

53.2 Maintainer documentation for the classes module, and ModuleElem

I shall suppose that the user documentation has already been read and digested. It could also be helpful to have read the documentation for `ring` (Sec.69) since the design philosophy here imitates that used for rings.

The class `module` is simply a reference counting smart pointer class to a concrete module (*i.e.* an object belonging to a class derived from `ModuleBase`).

A `ModuleElem`, like a `RingElem`, comprises two components: one specifying the algebraic structure to which the value belongs, and the other being an opaque representation of the value which can be correctly interpreted only by the owning module. The data members are:

```
module myM;             // the module to which the ModuleElem belongs
ModuleRawValue myValue; // "opaque" representation of the value,
                        // concrete modules must "import" this value.
```

The design philosophy for modules follows closely that used for rings. This means that every operation on `ModuleElems` is actually effected by calling the appropriate member function of the owning `module`. These member functions expect raw values as input. A normal `ModuleElem` stores within itself both the identity of the `module` to which it belongs and its value as an element of that particular module – we call the first datum the `owner` and the second datum the `RawValue`. A RawValue can be correctly interpreted only if supplied as argument to a member function of the owning module – calling module member functions for an incompatible concrete module and RawValue will very likely have grave consequences (officially stated as *undefined behaviour*, and most probably perceived as a program crash).

The member functions of a module **do not check** their arguments for being sensible. This decision is largely just a design policy imitating that used for rings, but may also lead to some slight beneficial effect on run-time performance. It does naturally imply that the programmer bears a considerable burden of responsibility.

(2.1) Member functions for operations on raw values [IGNORE THIS – OUT OF DATE]

For ring elements (especially those in a small finite field), noticeable speed gains arise from using directly raw values and ring member functions. For modules the analogous effect exists in theory but will likely be negligible in practice. Nevertheless we list here the member functions of a module; this list will be useful to library authors who wish to create their own concrete module classes.

Let `v` be a non-const RawValue, and `v1, v2` const RawValues belonging to `M`. Let `r` be a RingBase::RawValue belonging to the base ring of `M`.

```
M.myNumCompts()
M.myBaseRing()
M.myGens()        -- returns a const ref to a C++ vector of module:elems
M.myZero()        -- returns a const ref to a ModuleElem

M.myNew(v)       -- allocates resources, apply only to uninitialized RawValue
M.myNew(v, v1)   -- allocates resources, apply only to uninitialized RawValue
M.myDelete(v)    -- releases resources
M.mySwap(v, w)
M.myAssign(v, v1)
M.myNegate(v, v1)
M.myAdd(v, v1, v2)
M.mySub(v, v1, v2)
M.myMul(v, r, v1)
M.myDiv(v, r, v1) -- NOTE funny arg order!
M.myOutput(out, v1)
M.myOutputSelf(out)
M.myIsZero(v1)
M.myIsEqual(v1, v2)
```

53.3 Bugs, Shortcomings and other ideas

This code is too new, largely untried/untested. As soon as it gets some use, there will be some material to put here :-)

The documentation is very incomplete. Will be fixed (eventually). Maintainer documentation is incompleter than user doc.

54 ModuleTermOrdering (Anna Bigatti)

54.1 User documentation for ModuleTermOrdering

An object of the class ModuleTermOrdering represents an ordering on the module monoid of module terms, i.e. such that the ordering respects the operation .... In CoCoALib orderings and gradings are intimately linked (for gradings see also `degree` (Sec.17) and `PPOrdering` (Sec.60)).

Currently, the most typical use for a ModuleTermOrdering object is as a constructor argument to a concrete `FreeModule` (Sec.33). At the moment there are ? functions which create new `ModuleTermOrderings`:

Pseudo-constructors: (where `PPO` is a `PPOrdering` (Sec.60), `shifts` is a `vector<degree>`, `perm` is `std::vector<long>`, `NumComponents` is a `long`)

```
NewWDegTOPos(PPO, NumComponents);
NewPosWDegTO(PPO, NumComponents);
NewWDegPosTO(PPO, NumComponents);
NewWDegTOPos(PPO, shifts);
NewWDegPosTO(PPO, shifts);
NewPosWDegTO(PPO, shifts);
NewWDegTOPos(PPO, perm);
NewWDegPosTO(PPO, perm);
NewWDegTOPos(PPO, shifts, perm);
NewWDegPosTO(PPO, shifts, perm);
```

where

```
WDeg is the degree (incl. the shifts)
TO   is the PPOrdering (incl. the degree, i.e. the first GrDim rows)
Pos  is the position (according to the "score" given by perm [NYI])
```

54.1.1 Example

```
P = Q[x,y] with StdDegLex (==> GradingDim = 1)
P(-2) (+) P(-1)  i.e. P^2 with shifts = [(2), (1)], and WDegTOPos
v1 = [x,0],   v2 = [0,y^2]:
WDeg(v1) = WDeg(x)+2 = 3,  WDeg(v2) = WDeg(y^2)+1 = 3
x < y^2 according to StdDegLex  (NB: not "Lex"!)
so v1 < v2
```

The operations on a ModuleTermOrdering object are:

```
out << MTO;      // output the MTO object to channel out
const std::vector<degree>& shifts(const ModuleTermOrdering& O);
long NumComponents(const ModuleTermOrdering& MTO);
long GradingDim(const ModuleTermOrdering& MTO);
const PPOrdering& ModPPOrdering(const ModuleTermOrdering& MTO);

bool IsWDegTOPos(const ModuleTermOrdering& MTO);// true iff MTO is implemented as WDegTOPos
bool IsPosWDegTO(const ModuleTermOrdering& MTO);
bool IsWDegPosTO(const ModuleTermOrdering& MTO);
```

output and OpenMath output is still questionable.

54.2 Maintainer documentation for ModuleTermOrdering

The general ideas behind the implementations of ModuleTermOrdering and ModuleTermOrderingBase are analogous to those used for ring and RingBase. ModuleTermOrdering is a simple reference counting smart-pointer class, while ModuleTermOrderingBase hosts the intrusive reference count (so that every concrete derived class will inherit it). See

The only remaining observation to make about the simple class ModuleTermOrdering is that I have chosen to disable assignment – I find it hard to imagine when it could be useful to be able to assign ModuleTermOrderings, and suspect that allowing assignment is more likely to lead to confusion and poor programming style.

There are ? concrete ModuleTermOrderings in the namespace CoCoA::MTO. The implementations are all simple and straightforward except for the matrix ordering which is a little longer and messier but still easy enough to follow.

See also the CoCoAReport "Free Modules".

54.3 Bugs, shortcomings and other ideas

54.3.1 do we need a class "shifts"?

---

55 NumTheory (John Abbott)

55.1 User documentation

55.1.1 Generalities

The functions in the `NumTheory` file are predominantly basic operations from number theory. Most of the functions may be applied to machine integers or big integers (*i.e.* values of type `BigInt` (Sec.8)). Please recall that computational number theory is not the primary remit of CoCoALib, so do not expect to find a complete collection of operations here – you would do better to look at Victor Shoup's NTL (Number Theory Library), or PARI/GP, or some other specialized library/system.

55.1.2 Examples

- ex-NumTheory1.C

55.1.3 The Functions Available For Use

Several of these functions give errors if they are handed unsuitable values: unless otherwise indicated below the error is of type `ERR::BadArg`. All functions expecting a modulus will throw an error if the modulus is less than 2 (or an `unsigned long` value too large to fit into a `long`).

The main functions available are:

- `gcd(m,n)` computes the non-negative gcd of `m` and `n`. If both args are machine integers, the result is of type `long` (or error if it does not fit); otherwise result is of type `BigInt` (Sec.8).
- `ExtGcd(a,b,m,n)` computes the non-negative gcd of `m` and `n`; also sets `a` and `b` so that `gcd = a*m+b*n`. If `m` and `n` are machine integers then `a` and `b` must be of type (signed) `long`. If `m` and `n` are of type `BigInt` (Sec.8) then `a` and `b` must also be of type `BigInt` (Sec.8).
- `InvMod(r,m)` computes the least positive inverse of `r` modulo `m`; returns 0 **if the inverse does not exist**. Gives error if `m < 2`. Result is of type `long` if `m` is a machine integer; otherwise result is of type `BigInt` (Sec.8).
- `lcm(m,n)` computes the non-negative lcm of `m` and `n`. If both args are machine integers, the result is of type `long`; otherwise result is of type `BigInt` (Sec.8). Gives error `ERR::ArgTooBig` if the lcm of two machine integers is too large to fit into an `long`.
- `IsPrime(n)` tests the positive number `n` for primality (may be **very slow** for larger numbers). Gives error if `n <= 0`.
- `IsProbPrime(n)` tests the positive number `n` for primality (fairly fast for large numbers, but in very rare cases may falsely declare a number to be prime). Gives error if `n <= 0`.
- `IsProbPrime(n,iters)` tests the positive number `n` for primality; performs `iters` iterations of the Miller-Rabin test (default value is 25). Gives error if `n <= 0`.
- `NextPrime(n)` and `PrevPrime(n)` compute next or previous positive prime (fitting into a machine integer); returns 0 if none exists. Gives error if `n <= 0`.
- `NextProbPrime(N)` and `PrevProbPrime(N)` compute next or previous positive probable prime (uses `IsProbPrime`). Gives error if `N <= 0`.
- `SmoothFactor(n,limit)` finds small prime factors of `n` (up to & including the specified `limit`); result is a `factorization` object. Gives error if `limit` is not positive or too large to fit into a `long`.
- `factor(n)` finds the complete factorization of `n` (may be **very slow** for large numbers); NB **implementation incomplete**
- `EulerPhi(n)` computes Euler's *totient* function of the positive number `n` (*i.e.* the number of integers up to `n` which are coprime to `n`, or the degree of the `n`-th cyclotomic polynomial). Gives error if `n <= 0`.
- `PrimitiveRoot(p)` computes the least positive primitive root for the positive prime `p`. Gives error if `p` is not a positive prime. May be **very slow** for large `p` (because it must factorize `p-1`).
- `MultiplicativeOrder(res,mod)` computes multiplicative order of `res` modulo `mod`. Gives error if `mod < 2` or `gcd(res,mod)` is not 1.
- `PowerMod(base,exp,modulus)` computes `base` to the power `exp` modulo `modulus`; result is least non-negative residue. If `modulus` is a machine integer then the result is of type `long` (or error if it does not fit), otherwise the result is of type `BigInt` (Sec.8). Gives error if `modulus <= 1`. Gives `ERR::DivByZero` if `exp` is negative and `base` cannot be inverted. If `base` and `exp` are both zero, it produces 1.
- `SimplestBigRatBetween(A,B)` computes the simplest rational between `A` and `B`
- `BinomialRepr(N,r)` produces the repr of `N` as a sum of binomial coeffs with "denoms" `r, r-1, r-2, ...`

Continued Fractions

Several of these functions give errors if they are handed unsuitable values: unless otherwise indicated below the error is of type `ERR::BadArg`.

Recall that any real number has an expansion as a **continued fraction** (*e.g.* see Hardy & Wright for definition and many properties). This expansion is finite for any rational number. We adopt the following conventions which guarantee that the expansion is unique:

- the last partial quotient is greater than 1 (except for the expansion of integers $<= 1$)
- only the very first partial quotient may be non-positive.

For example, with these conventions the expansion of -7/3 is (-3, 1, 2).

The main functions available are:

- `ContFracIter(q)` constructs a new continued fraction iterator object
- `IsEnded(CFIter)` true iff the iterator has moved past the last *partial quotient*
- `IsFinal(CFIter)` true iff the iterator is at the last *partial quotient*
- `quot(CFIter)` gives the current *partial quotient* as a `BigInt` (Sec.8) (or throws `ERR::IterEnded`)
- `*CFIter` gives the current *partial quotient* as a `BigInt` (Sec.8) (or throws `ERR::IterEnded`)
- `++CFIter` moves to next *partial quotient* (or throws `ERR::IterEnded`)
- `ContFracApproximant()` for constructing a rational from its continued fraction quotients
- `CFA.myAppendQuot(q)` appends the quotient `q` to the continued fraction
- `CFA.myRational()` returns the rational associated to the continued fraction
- `CFApproximantsIter(q)` constructs a new continued fraction approximant iterator
- `IsEnded(CFAIter)` true iff the iterator has moved past the last "partial quotient"
- `*CFAIter` gives the current continued fraction approximant as a `BigRat` (Sec.9) (or throws `ERR::IterEnded`)
- `++CFAIter` moves to next approximant (or throws `ERR::IterEnded`)
- `CFApprox(q,eps)` gives the simplest cont. frac. approximant to `q` with relative error at most `eps`

Chinese Remaindering – Integer Reconstruction

CoCoALib offers the `CRTMill` to reconstruct an integer from several residue-modulus pairs via Chinese Remaindering. At the moment the moduli from distinct pairs must be coprime.

The operations available are:

- `CRTMill()` ctor; initially the residue is 0 and the modulus is 1
- `CRT.myAddInfo(res,mod)` give a new residue-modulus pair to the `CRTMill`
- `residue(CRT)` the combined residue with absolute value less than `modulus(CRT)`
- `modulus(CRT)` the product of the moduli of all pairs given to the mill

Rational Reconstruction

CoCoALib offers two heuristic methods for reconstructing rationals from residue-modulus pairs; they have the same user interface but internally one algorithm is based on continued fractions while the other uses lattice reduction. The methods are heuristic, so may (rarely) produce an incorrect result.

The operations available are:

- `RatReconstructByContFrac()` ctor for creating a reconstruction mill via continued fraction method
- `RatReconstructByContFrac(threshold)` ctor for continued fraction method mill with given threshold (0 –> use default)
- `RatReconstructByLattice()` ctor for creating a reconstruction mill via lattice method
- `RatReconstructByLattice(SafetyFactor)` ctor for lattice method mill with given SafetyFactor (0 –> use default)
- `reconstructor.myAddInfo(res,mod)` give a new residue-modulus pair to the reconstructor
- `IsConvincing(reconstructor)` gives `true` iff the mill can produce a *convincing* result
- `ReconstructedRat(reconstructor)` gives the reconstructed rational (or an error if `IsConvincing` is not true).

There is also a function for deterministic rational reconstruction which requires certain bounds to be given in input. It uses the continued fraction method.

- `RatReconstructWithBounds(e,P,Q,res,mod)` where `e` is upper bound for number of "bad" moduli, `P` and `Q` are upper bounds for numerator and denominator of the rational to be reconstructed, and `(res[i],mod[i])` is a residue-modulus pair with distinct moduli being coprime.

55.2 Maintainer Documentation

Correctness of `ExtendedEuclideanAlg` is not immediately clear, because the cofactor variables could conceivably overflow – in fact this cannot happen (at least on a binary computer): for a proof see Shoup's book *A Computational Introduction to Number Theory and Algebra*, in particular Theorem 4.3 and the comment immediately following it. There is just one line where a harmless "overflow" could occur – it is commented in the code.

Several functions are more complicated than you might expect because I wanted them to be correct for all possible machine integer inputs (*e.g.* including the most negative `long` value).

In some cases the function which does all the work is implemented as a file local function operating on `unsigned long` values: the function should normally be used only via the "dispatch" functions whose args are of type `MachineInt` (Sec.46) or `BigInt` (Sec.8).

The continued fractions functions are all pretty simple. The only tricky part is that the "end" of the `ContFracIter` is represented by both `myFrac` and `myQuot` being zero. This means that a newly created iterator for zero is already ended.

`CFApproximantsIter` delegates most of the work to `ContFracIter`.

55.3 Bugs, Shortcomings, etc.

Several functions return `long` values when perhaps `unsigned long` would possibly be better choice (since it offers a greater range, and in the case of `gcd` it would permit the fn to return a result always, rather than report "overflow"). The choice of return type was dictated by the coding conventions, which were in turn dictated by the risks of nasty surprises to unwary users unfamiliar with the foibles of unsigned values in C++.

Should there also be procedural forms of functions which return `BigInt` (Sec.8) values? (*e.g.* gcd, lcm, InvMod, PowerMod, and so on).

Certain implementations of `PowerMod` should be improved (*e.g.* to use `PowerModSmallModulus` whenever possible). Is behaviour for 0^0 correct?

`LucasTest` should produce a certificate, and be made publicly accessible.

How should the cont frac iterators be printed out???

`ContFracIter` could be rather more efficient for rationals having very large numerator and denominator. One way would be to compute with num and den divided by the same large factor (probably a power of 2), and taking care to monitor how accurate these "scaled" num and den are. I'll wait until there is a real need before implementing (as I expect it will turn out a bit messy).

`CFApproximantsIter::operator++()` should be made more efficient.

56 OpenMath (John Abbott)

56.1 User documentation for OpenMath

These files offer two types: `OpenMathOutput` for sending data in OpenMath format, and `OpenMathInput` for receiving data sent in OpenMath format. Since OpenMath specifies more than one encoding, you must specify which encoding is to be used when creating one of these input/output channels. Here are two (ugly) examples:

```
OpenMathOutput OMOut(new OpenMathOutputXML(cout));
OpenMathInput OMIn(new OpenMathInputXML(cin));
```

These commands say that the XML encoding is to be used, and that `cin`/`cout` as the data transport mediums. Once created, these OpenMath i/o channels can be used analogously to the standard C++ i/o channels.

56.2 Maintainer documentation for OpenMath

`OpenMathInput` and `OpenMathOutput` use the template class `SmartPtrIRC` (Sec.84) as their implementations. They are reference counting "smart pointers" (but I'm not sure why, perhaps just for simplicity?).

There are six different `operator<<` for built in integer types because I needed at least two (one for `long` and one for `unsigned long`) and the compiler complained about ambiguities for other integral types because it could have converted equally well to either `long` or `unsigned long`. There are only two corresponding member functions, as the implementations of `operator<<` cast to either `long` or `unsigned long`.

56.3 Bugs, Shortcomings and other ideas

Use `boost::shared_ptr` instead of `SmartPtrIRC` (Sec.84)?

Documentation woefully incomplete. Actually the whole implementation needs a thorough revision, perhaps in collaboration with some others who are attemtping to implement OpenMath.

Code written hastily, so incomplete, largely untested, does not follow the coding standards (esp. member fn names).

Need a safer way to send "brackets" (e.g. OpenMath apply begin and end tokens).

Should `OpenMathSymbol` have ctors with one `string` and one `char*`?

OpenMath attributes completely ignored.

57 OrdvArith (John Abbott)

57.1 User documentation for OrdvArith

`OrdvArith` objects are "low level" values, and thus probably of little interest to most users of CoCoALib. They are compressed vectors of non-negative small integers, and were created to represent (indirectly) the exponent vectors of power products in a way which allows fast comparison of two power products (using the PP ordering for the polynomial ring).

The main operations on `OrdvArith` values are comparison and multiplication (which entails simply summing the vectors) – these are designed to be fast. There are also conversions to and from true exponent vectors, and sundry other operations.

The overall design mimicks that of `ring`s and `RingElem`s but with a less convenient interface. In fact, all operations on an `OrdvElem` must be effected through an explicit member function call.

57.1.1 Further features of PPOrdering for CoCoA library developers

There are many more operations on a PPOrdering object than those listed above. These further operations are used in the implementation of some sorts of PPMonoid, and some sorts of (distributed multivariate) polynomial. Run-time efficiency is the primary reason for the existence of these functions: so be warned that cleanliness is sometimes sacrificed at the altar of speed.

To better understand the what and the why of a PPOrdering, let us begin by setting the scene. We recall that for all practical purposes an arithmetic ordering on power products can be specified by a matrix of integers M as follows:

> Let t1 = x_1^e_1 * x_2^e_2 * ... * x_n^e_n be a power product, and t2 = x_1^f_1 * x_2^f_2 * ... * x_n^f_n be another. Then we call (e_1, e_2,..., e_n) the **exponent vector** for t1, and similarly for t2. For brevity we shall write expv(t1), etc.

> The matrix M determines the ordering thus: we say that t1 < t2 iff M*expv(t1) comes before M*expv(t2) in lex ordering. We call the product M*expv(t1) the **order vector** for t1, and for brevity we shall write ordv(t1) to denote it.

Typically the matrix M is subject to some suitability criteria, e.g. M should be square and invertible. We shall assume henceforth that M has been chosen so that all order vectors contain only non-negative entries. While reading the rest of these notes it may be convenient to think of M as being non-singular, so that there is a 1-1 correspondence between power products and their order vectors.

Now the scene has been set, we can explain what a PPOrdering object does. Abstractly, we can think of the PPOrdering as embodying the matrix M; in practice shortcuts are taken if M has a recognised special structure. A PPOrdering offers a number of simple operations on order vectors including conversion to and from an exponent vector.

Before listing the operations a PPOrdering offers, let me emphasise that a PPOrdering NEVER allocates memory: the caller must always supply a pointer to enough free memory (this is relevant mainly for conversion between order vectors and exponent vectors). An exponent vector is taken to be a C array of NumIndets(PPO) entries each of type PPOrdering::ExpvElem where the i-th entry is the i-th exponent. An order vector is taken to be a C array of OrdvWords(PPO) entries each of type PPOrdering::OrdvElem – the entries do not admit an easy interpretation.

```
OrdvWords(PPO)                  -- "size" that an order vector must have
                                   (see note in text above)
PPO->myInit(ordv)               -- initialize an order vector to zero
PPO->myInitFromExpv(ordv, expv) -- initialize an order vector from an exponent vector
PPO->myAssign(l_ordv, r_ordv)   -- assign r_ordv to l_ordv
PPO->myMul(ordv, ordv1, ordv2)  -- ordv = ordv1 * ordv2
PPO->myDiv(ordv, ordv1, ordv2)  -- ordv = ordv1 / ordv2 (quotient MUST exist)
PPO->myMulIndetPower(ordv, var, exp) -- ordv *= var^exp
PPO->myCmp(l_ordv, r_ordv)      -- return -1,0,+1 according as l_ordv <,=,> r_ordv
PPO->myCmpExpvs(l_expv, r_expv) -- return -1,0,+1 according as l_expv <,=,> r_expv
PPO->myComputeExpv(expv, ordv)  -- convert ordv into expv
PPO->myLog(ordv, var)           -- return power of var in PP corresponding to ordv (result is an unsigned
PPO->myDeg(d, ordv)             -- compute degree of ordv, putting result in d
PPO->myCmpDeg(ordv1, ordv2)     -- return -1,0,+1 according as deg(ordv1) <,=,> deg(ordv2)
PPO->myIsOne(ordv)              -- return true iff ordv corr to the power product 1
PPO->myOutput(OMOut, ordv)      -- output ordv on an OpenMath channel
```

NOTES

- (1) since order vectors are linearly related to exponent vectors, the functions myMul and myDiv actually compute the sum or difference of the order vectors. myDiv will give a junk answer if the PP corresponding to ordv2 does not divide that corresponding to ordv1.
- (2) A useful trick for "converting" a C++ vector<> to a C vector is to pass the address of the zeroth element of the C++ vector<> (thus: &vec[0]).
- (3) myCmpExpvs is likely to be rather slower than myCmp (even with "lex"), but possibly faster than converting both expvs to ordvs and then calling myCmp.

- (4) The two functions which convert between expv and ordv representations might be quite slow, especially if a general ordering is used. Even with the simplest ordering (i.e. lex) the conversion is not instant because order vectors are held in a packed representation.

57.1.2 To Write a New Concrete PPOrdering Class

If you plan to write a new concrete class derived from PPOrderingBase then you must know about these virtual member functions:

```
myMulSpecial(ordv, ordv1, ordv2)
myDivSpecial(ordv, ordv1, ordv2)
myCmpSpecial(l_ordv, r_ordv)
```

Their existence and their curious names arise from the need to make myMul, myDiv and myCmp inline for the common case of densely represented order vectors. If you plan to add a new concrete PPOrdering which is based on densely represented order vectors then have a look at the implementations in the PPOrderingDense files.

If you are considering implementing some other sort of concrete PPOrdering, read on. The inline functions myMul, myDiv and myCmp use the value of the data member PPOrderingBase::myOrdvWordsForCmp to decide what to do: if the value is non-zero, they assume a dense representation is being used and execute the corresponding loop, otherwise they call corresponding "special" virtual function to do all the work. You will probably need to set myOrdvWordsForCmp to zero, and you must place your implementations for product, quotient and comparison in the "special" functions.

[I know this is ugly, but cannot see how else to guarantee that the dense case can be compiled inline]

57.2 Maintainer documentation for OrdvArith

The implementation of `PPOrderingBase` is closely tied to those of `PPMonoid` (Sec.58) and `DistrMPolyInlPP` (Sec.22). Be aware of this if you want to change `PPOrderingBase` at all.

On the whole, `PPOrderingBase` is a fairly straightforward abstract class: - it has an intrusive reference count; - it defines numerous pure virtual functions; - it has four data members.

The main oddities and other points of note are below.

Data member `myNumIndets` is required when dealing with exponent vectors (since C vectors do not record their own length). It is the number of valid entries in a C vector representing an exponent vector.

Data member myGradingDim specifies how many initial components of an order vector comprise the grading. It is needed in myDeg.

Data member myOrdvWords is used only to supply the return value to the friend function OrdvWords. This value is needed so that a caller can allocate the correct amount of space in which to build a new order vector value. By default this is initialized to a huge value, so that it will quickly become evident at run-time if it hasn't been initialized to a sane value.

Data member myOrdvWordsForCmp is used in myMul, myDiv and myCmp to choose between an inline function and a virtual call. Its value may be non-zero and different from myOrdvWords if a redundant representation is being used (e.g. for a DegRevLex ordering). By default this is initialized to a huge value, so that it will quickly become evident at run-time if it hasn't been initialized to a sane value.

The member functions myMul, myDiv, and myCmp are non-virtual so that the compiler can implement them inline: at run-tme they check the data member myOrdvWordsForCmp to decide whether to the use the inline function or delegate to a "shadow" virtual function. This rather ugly arrangement was necessary to achieve acceptable run-time performance.

The member function myMulIndetPower is not pure because a reasonable generic implementation exists. Similarly, myOutput(OMOut, ordv) is not pure.

57.3 Bugs, Shortcomings and other ideas

This documentation has been salvaged from an old version of PPOrdering.txt. It probably needs major modification.

In some ways, myCmp could simply be operator(); thus calls would look like ord(ordv1, ordv2) where ord is an object of type PPOrdering.

Are the typedefs PPOrdering::OrdvElem and PPOrdering::ExpvElem really independent?

Why is myOrdvBuffer a C++ vector<> instead just a C vector???

We need a way to handle order vectors which have large integer entries! (also ordering matrices with large integer entries). Recall that some ordvs may involve mpz_t integers! Note that the polynomial type needs to know how big an ordv can be: that's what the OrdvWords member function is for.

Should "degrevlex" actually store an extra component so that deg(...,x[0]) can be calculated easily? Do we really need this to be quick? It would be needed for computing GCDs, testing divisibility etc, but these operations would normally be done only on "rich PP" objects – talk to Anna!

The restriction to order compatible gradings may not be wholly necessary. The PPs in a polynomial homogeneous with respect to a k-dimensional grading are completely specified by n-k of the entries in the order vector, though precisely which entries must be retained depends on the grading and the ordering. Thus a later generalization to non order compatible gradings may not be too painful.

ANNA: must add a section about modular order matrix JOHN: yes, you must! Where does 46336 come from???

The default implementation of myIsIndet is not very efficient, but is it really worth writing many different (efficient) implementations?

58 PPMonoid (John Abbott)

58.1 User documentation for the classes PPMonoid, PPMonoidElem and PPMonoidBase

The classes `PPMonoid` and `PPMonoidElem` are analogous to `ring` (Sec.69) and `RingElem`. A `PPMonoid` represents a (multiplicative) power product monoid with grading and compatible total arithmetic ordering; a `PPMonoidElem` represents an element of a `PPMonoid`, *i.e.* a power product.

`PPMonoid` and `PPMonoidElem` are used inside the implementation of `SparsePolyRing` (Sec.87) (multivariate polynomial rings).

You do not have to deal directly with `PPMonoid` unless you want to work solely with power-products, or use some particular implementation for a specific need in your `SparsePolyRing` (Sec.87) – *e.g.* huge exponents, very sparse power-products, fast ordering or fast access to exponents.

The implementations of `PPMonoid`s are optimized for different uses:

- `PPMonoidEv`: stores the *Exponent vector*; it is good for accessing the exponents, but slow for ordering; with optional 3rd arg `BigExps` the exponents are stored as `BigInt` (Sec.8)'s
- `PPMonoidOv`: stores the *Order vector*; it is good for ordering, but slow for accessing the exponents
- `PPMonoidEvOv`: stores the *Exponent vector* and the *Order vector*; it is good for accessing the exponents and for ordering but uses more memory and takes more time to assign.

58.1.1 Examples

- ex-PPMonoidElem1.C
- ex-PPMonoidElem2.C

58.1.2 Operations PPMonoids

Recall that every `PPMonoid` is graded, and has a degree-compatible total arithmetical ordering; the grading and ordering must be specified when the `PPMonoid` is created. For convenient input and output, also the names of the indeterminates generating the monoid must be specified when the monoid is created.

If you expect to use large exponents then you should use only the special `PPMonoid` created by `PPMonoidBigEv`. The other `PPMonoid`s should usually be fine for exponents up to 1000 or more; the true limit depends on the specific monoid, the number of indeterminates, and the `PPOrdering` (Sec.60). At the moment there is no way to find out what the true limit is (see *Bugs* section), and no warning is given should the limit be exceeded: you just get a wrong answer.

Pseudo-constructors of PPMonoid

To create a `PPMonoid` use the function `NewPPMonoid` (the default currently chooses `PPMonoidEv`). To create a `PPMonoid` object of a specific type use one of the pseudo-constructors related to the concrete monoid classes:

Given `PPO` a `PPOrdering` (Sec.60) or `PPOrderingCtor` (*i.e.* `lex`, `StdDegLex`, or `StdDegRevLex`), and `IndetNames` a `vector` of `symbol` (Sec.90)

- `NewPPMonoid(IndetNames, PPO)` – same as `NewPPMonoidEv`
- `NewPPMonoidEv(IndetNames, PPO)`
- `NewPPMonoidEv(IndetNames, PPO, BigExps)` – `BigExps` is just an enum member.
- `NewPPMonoidOv(IndetNames, PPO)`
- `NewPPMonoidEvOv(IndetNames, PPO)`

Operations

- `cout << PPM` – print `PPM` on `cout`
- `NumIndets(PPM)` – number of indeterminates
- `ordering(PPM)` – the `PPOrdering` (Sec.60) inherent in `PPM`
- `GradingDim(PPM)` – the dimension of the grading (zero if ungraded)
- `symbols(PPM)` – `std::vector` of the `symbol` (Sec.90)s in `PPM` (*i.e.* names of the indets)
- `IndetSymbol(PPM, i)` – the `symbol` (Sec.90) for the `i`-th indeterminate
- `PPM1 == PPM2` – true iff `PPM1` and `PPM2` are identical (*i.e.* same addr)
- `PPM1 != PPM2` – true unless `PPM1` and `PPM2` are identical

These pseudo-constructors are described in the section about `PPMonoidElem`s

- `one(PPM)`
- `indet(PPM, i)`
- `IndetPower(PPM, i, exp)`
- `indets(PPM)`

58.1.3 Summary of functions for PPMonoidElems

See also some example programs in the CoCoALib/examples/ directory.

When a new object of type `PPMonoidElem` is created the monoid to which it belongs must be specified either explicitly as a constructor argument, or implicitly as the monoid associated with some constructor argument. Once the `PPMonoidElem` object has been created it is not possible to make it belong to any other monoid. Comparison and arithmetic between objects of type `PPMonoidElem` is permitted only if they belong to the same identical monoid.

Note: when writing a function which has an argument of type `PPMonoidElem`, you should specify the argument type as `ConstRefPPMonoidElem`, or `RefPPMonoidElem` if you want to modify its value.

Let `PPM` be a `PPMonoid`; for convenience, in comments we shall use x[i] to refer to the i-th indeterminate in PPM. Let `pp` be a non-const `PPMonoidElem`, and `pp1` and `pp2` be `const PPMonoidElems` (all belonging to PPM). Let `expv` be a `vector<long>` of size equal to the number of indeterminates.

- `PPMonoidElem t(PPM)` – create new PP in `PPM`, value is 1
- `PPMonoidElem t(PPM, expv)` – create new PP in `PPM`, value is product x[i]^expv[i]
- `PPMonoidElem t(pp1)` – create a new copy of `pp1`, belongs to same PPMonoid as `pp1`

- `one(PPM)` – the 1 belonging to `PPM`
- `indet(PPM, i)` – create a new copy of x[i] the i-th indeterminate of `PPM`
- `IndetPower(PPM, i, n)` – create x[i]^n, `n`-th power of `i`-th indeterminate of `PPM`
- `indets(PPM)` – `std::vector` (reference) whose n-th entry is n-th indet as a `PPMonoidElem`
- `owner(pp1)` – returns the `PPMonoid` to which `pp1` belongs
- `IsOne(pp1)` – returns true iff `pp1` = 1
- `IsIndet(i, pp1)` – returns true iff `pp1` is an indet; if true, puts index of indet into `i`
- `IsIndetPosPower(i, N, pp1)` – returns true iff `pp1` is a positive power of some indet; when the result is true (signed long) `i` and (`BigInt` (Sec.8)) `N` are set so that `pp1 == IndetPower(owner(pp), i, N);` (otherwise unchanged) if `pp1` == 1 then the function throws `ERR::BadArg`
- `IsIndetPosPower(i, n, pp1)` – same as above, where `n` is long
- `cmp(pp1, pp2)` – compare `pp1` with `pp2` using inherent ordering; result is integer <0 if `pp1 < pp2`, =0 if `pp1 == pp2`, and >0 if `pp1 > pp2`
- `pp1 == pp2` – the six standard comparison operators...
- `pp1 != pp2` – ...
- `pp1 < pp2` – ... (inequalities use the ordering inherent in `PPM`)
- `pp1 <= pp2` – ...
- `pp1 > pp2` – ...
- `pp1 >= pp2` – ...
- `pp1 * pp2` – product of `pp1` and `pp2`
- `pp1 / pp2` – quotient of `pp1` by `pp2`, quotient **must** be exact (see the function `IsDivisible` below)
- `colon(pp1, pp2)` – *colon quotient* of `pp1` by `pp2`, *i.e.* `pp1/gcd(pp1,pp2)`
- `gcd(pp1, pp2)` – gcd of `pp1` and `pp2`
- `lcm(pp1, pp2)` – lcm of `pp1` and `pp2`
- `radical(pp1)` – radical of `pp1`
- `power(pp1, n)` – `n`-th power of `pp1` (NB: you **cannot** use `pp1^n`, see below)
- `IsCoprime(pp1, pp2)` – tests whether `pp1` and `pp2` are coprime
- `IsDivisible(pp1, pp2)` – tests whether `pp1` is divisible by `pp2`
- `IsRadical(pp1)` – test whether `pp1` is radical, *i.e.* if `pp1 == radical(pp1)`
- `AssignOne(pp)` – sets `pp = 1`
- `swap(pp, pp_other)` – swaps the values of `pp` and `pp_other`
- `pp = pp1` – assignment (`pp` and `pp1` must belong to same PPMonoid)
- `pp *= pp1` – same as `pp = pp * pp1`
- `pp /= pp1` – same as `pp = pp / pp1`
- `StdDeg(pp1)` – standard degree of `pp1`; result is of type `long`
- `wdeg(pp1)` – weighted degree of `pp1` (using specified grading); result is of type `degree` (Sec.17)
- `CmpWDeg(pp1, pp2)` – result is integer <0 =0 >0 according as `wdeg(pp1) < = > wdeg(pp2)`; order on weighted degrees is lex, see `degree` (Sec.17)
- `CmpWDegPartial(pp1, pp2, i)` – result is integer <0 =0 >0 as `CmpWDeg` wrt the first `i` components of the weighted degree

- `exponent(pp1, i)` – exponent of x[i] in `pp1` (result is a `long`)
- `BigExponent(pp1, i)` – exponent of x[i] in `pp1` (result is a `BigInt` (Sec.8))
- `exponents(expv, pp)` – fills vector (of long) `expv` so that `expv[i] = exponent(pp, i)` for i=0,..,NumIndets(PPM)-1
- `BigExponents(expv, pp)` – fills vector (of BigInt) `expv` so that `expv[i] = BigExponent(pp, i)` for i=0,..,NumIndets(PPM)-1
- `cout << pp1` – print out the value of `pp1`

58.2 Library Contributor Documentation

This section comprises two parts: the first is about creating a new type of PP monoid; the second comments about calling the member functions of `PPMonoidBase` directly.

58.2.1 To add a new type of concrete PPMonoid class

My first suggestion is to look at the code implementing `PPMonoidEv`. This is a simple PP monoid implementation: the values are represented as C arrays of exponents. Initially you should ignore the class `CmpBase` and those derived from it; they are simply to permit fast comparison of PPs in certain special cases.

First, a note about "philosophy". As far as we can tell, the programming language C++ does not have a built-in type system sufficiently flexible (and efficient) for our needs, consequently we have to build our own type system on top of what C++ offers. The way we have chosen to do this is as follows (note that the overall scheme used here is similar to that used for rings and their elements).

To fit into CoCoALib your new class must be derived from `PPMonoidBase`. Remember that any operation on elements of your PP monoid will be effected by calling a member function of your new monoid class.

The monoid must be a cartesian power of N, the natural numbers, with the monoid operation (called "multiplication") being vector addition – the vector should be thought of as the vector of exponents in a power product. The monoid must have a total arithmetic ordering; often this will be specified when the monoid is created. The class `PPOrdering` (Sec.60) represents the possible orderings.

Here is a summary of the member functions which must be implemented. All the functions may be called for a **const** `PPMonoid`, for brevity the `const` qualifier is omitted. I use two abbreviations:

`RawPP`	is short for `PPMonoidElemRawPtr`
`ConstRawPP`	is short for `PPMonoidElemConstRawPtr`

Note: all arithmetic functions must tolerate argument aliasing (*i.e.* any pair of arguments may be identical).

Constructors: these all allocate memory which must eventually be freed (by calling `myDelete`); the result is a pointer to the memory allocated.

- `PPMonoidElemRawPtr PPMonoidBase::myNew()` – initialize pp to the identity
- `PPMonoidElemRawPtr PPMonoidBase::myNew(const vector<int>& expv)` – initialize pp from exponent vector `expv`
- `PPMonoidElemRawPtr PPMonoidBase::myNew(const RawPP& pp1)` – initialize pp from `pp1`

Destructor: there is only one of these, its argument must be initialized

- `void PPMonoidBase::myDelete(PPMonoidElemRawPtr pp)` – destroy `pp`, frees memory

Assignment etc:

- `void PPMonoidBase::mySwap(RawPP pp1, RawPP pp2)` – swap the values of `pp1` and `pp2`
- `void PPMonoidBase::myAssign(RawPP pp, ConstRawPP pp1)` – assign the value of `pp1` to `pp`
- `void PPMonoidBase::myAssign(RawPP pp, const vector<int>& expv)` – assign to `pp` the PP with exponent vector `expv`

Arithmetic: in all cases the first arg is where the answer is placed, aliasing is permitted (*i.e.* arguments need not be distinct); `myDiv` result is **undefined** if the quotient does not exist!

- `const PPMonoidElem& myOne()` – reference to 1 in the monoid
- `void myMul(RawPP pp, ConstRawPP pp1, ConstRawPP pp2)` – effects pp = pp1*pp2
- `void myMulIndetPower(RawPtr pp, long i, unsigned long exp)` – effects pp *= indet(i)^exp
- `void myDiv(RawPP pp, ConstRawPP pp1, ConstRawPP pp2)` – effects pp = pp1/pp2 (if it exists)
- `void myColon(RawPP pp, ConstRawPP pp1, Const RawPP pp2)` – effects pp = pp1/gcd(pp1,pp2)
- `void myGcd(RawPP pp, ConstRawPP pp1, ConstRawPP pp2)` – effects pp = gcd(pp1, pp2)
- `void myLcm(RawPP pp, ConstRawPP pp1, ConstRawPP pp2)` – effects pp = lcm(pp1, pp2)
- `void myPower(RawPP pp, ConstRawPP pp1, int exp)` – effects pp = pp1^exp

Comparison and testing: each PP monoid has associated with it a **term ordering**, *i.e.* a total ordering which respects the monoid operation (multiplication)

- `bool myIsCoprime(ConstRawPP pp1, ConstRawPP pp2)` – true iff gcd(pp1, pp2) is 1
- `bool myIsDivisible(ConstRawPP t1, ConstRawPP t2)` – true iff t1 is divisible by t2
- `int myCmp(ConstRawPP t1, ConstRawPP t2)` – result is <0, $=0$, >0 according as t1 $<,=,>$ t2
- NYI `int myHomogCmp(ConstRawPP t1, ConstRawPP t2)` – as cmp, but assumes t1 and t2 have the same degree

Sundries:

- `degree myDeg(ConstRawPP t)` – total degree
- `long myExponent(ConstRawPtr rawpp, long i)` – exponent of i-th indet in pp
- `void myBigExponent(BigInt& EXP, ConstRawPtr rawpp, long i)` – EXP = degree of i-th indet in pp
- `void myExponents(vector<long>& expv, ConstRawPP t)` – get exponents, put them in expv
- `void myBigExponents(vector<BigInt>& expv, ConstRawPP t)` – get exponents, put them in expv
- `ostream& myOutput(ostream& out, const RawPP& t)` – prints t on out; default defn in PPMonoid.C

Query functions:

- `long myNumIndets()` – number of indeterminates generating the monoid
- `const symbol& myIndetName(long var)` – name of indet with index var

58.2.2 To add a new member function to PPMonoidBase

You will have to edit `PPMonoid.H` and possibly `PPMonoid.C` (*e.g.* if there is to be a default definition). Arguments representing PPs should be of type `RawPP` if they may be modified, or of type `ConstRawPP` if they are read-only. See also the Coding Conventions about names of member functions.

If you do add a new pure virtual member function, you will have to add definitions to all the existing concrete PP monoid classes (otherwise they will become uninstantiable). Don't forget to update the documentation too!

58.2.3 Calculating directly with raw PPs

Values of type `PPMonoidElem` are intended to be simple and safe to use but with some performance penalty. There is also a "fast, ugly, unsafe" option which we shall describe here.

The most important fact to heed is that a `PPMonoidElemRawPtr` value is **not** a C++ object – it does not generally know enough about itself even to destroy itself. This places a considerable responsibility on the programmer, and probably makes it difficult to write exception clean code. You really must view the performance issue as paramount if you plan to use raw PPs! In any case the gain in speed will likely be only slight.

The model for creation/destruction and use of raw PPs is as follows: (NB see *Bugs* section about exception-safety) - (1) an uninitialized raw PP is acquired from the system; - (2) the raw PP is initialized by calling an initialization function (typically called `myNew`) – this will generally acquire further resources; - (3) now the RawPP may be used for i/o, arithmetic, and so forth; - (4) finally, when the value is no longer required the extra resources acquired during initialization should be released by calling the `myDelete` function – failure to call `myDelete` will probably result in a memory leak.

Here is some pseudo C++ code to give an idea

```
const PPMonoid& M = ...; // A PPMonoid from somewhere

PPMonoidElemRawPtr t;    // A wrapped opaque pointer; initially points into hyperspace.

t = M->myNew();          // Allocate resources for a new PP belonging to M;
                         // there are two other myNew functions.
.... operations on t; always via a member function of the monoid M ...

M->myDelete(t);          // "destroy" the value t held; t points into hyperspace again.
```

NOTE: the only functions which take a pointer into hyperspace are `PPMonoidBase::myNew`; many functions, *e.g.* `PPMonoidBase::myMul`, write their result into the first argument and require that that first argument be already allocated/initialized.

NOTE: if an exception is thrown after `M->myNew` and before `M->myDelete` then there will be a memory leak (unless you correctly add a `try...catch` block). If `t` is just to hold a temporary local value then it is better to create a full `PPMonoidElem` and then let `t` be its `RawPtr`; this should avoid memory leaks.

58.3 Maintainer documentation for PPMonoid, PPMonoidElem, and PPMonoidBase

The general structure here mirrors that of rings and their elements, so you may find it helpful to read ring.txt if the following seems too opaque. At first sight the design may seem complex (because it comprises several classes), but there's no need to be afraid.

The class `PPMonoid` is a reference counting smart pointer to an object derived from `PPMonoidBase`. This means that making copies of a `PPMonoid` is very cheap, and that it is easy to tell if two `PPMonoid`s are identical. Assignment of `PPMonoid`s is disabled because I am not sure whether it is useful/meaningful. `operator->` allows member functions of `PPMonoidBase` to be called using a simple syntax.

The class `PPMonoidBase` is what specifies the class interface for each concrete PP monoid implementation, i.e. the operations that it must offer. It includes an intrusive reference count for compatibility with `PPMonoid`. Since it is inconceivable to have a PP monoid without an ordering, there is a data member for memorizing the inherent `PPOrdering`. This data member is `protected` so that it is accessible only to friends and derived classes.

The function `PPMonoidBase::myOutput` for printing PPs has a reasonable default definition.

The situation for elements of a PP monoid could easily appear horrendously complicated. The basic idea is that a PP monoid element comprises two components: one indicating the `PPMonoid` to which the value belongs, and the other indicating the actual value. This allows the user to employ a notationally convenient syntax for many operations – the emphasis is on notational convenience rather than ultimate run-time efficiency.

For an element of a PP monoid, the owning `PPMonoid` is specified during creation and remains fixed throughout the life of the object; in contrast the value may be varied (if C++ const rules permit). The value is indicated by an opaque pointer (essentially a wrapped `void*`): only the owning `PPMonoid` knows how to interpret the data pointed to, and so all operations on the value are effected by member functions of the owning `PPMonoid`.

I do not like the idea of having naked `void*` values in programs: it is too easy to get confused about what is pointing to what. Since the value part of a `PPMonoidElem` is an opaque pointer (morally a `void*`), I chose to wrap

it in a lightweight class; actually there are two classes depending on whether the pointed to value is `const` or not. These classes are `PPMonoidElemRawPtr` and `PPMonoidElemConstRawPtr`; they are opaque pointers pointing to a value belonging to some concrete PP monoid (someone else must keep track of precisely which PP monoid is the owner).

The constructors for `PPMonoidElemRawPtr` and `PPMonoidElemConstRawPtr` are `explicit` to avoid potentially risky automatic conversion of any old pointer into one of these types. The naked pointer may be accessed via the member functions `myRawPtr`. Only implementors of new PP monoid classes are likely to find these two opaque pointer classes useful.

I now return to the classes for representing fully qualified PPs. There are three very similar yet distinct classes for elements of PP monoids; the distinction is to keep track of constness and ownership. I have used inheritance to allow natural automatic conversion among these three classes (analogously to `RingElem`, `ConstRefRingElem`)

- A `PPMonoidElem` is the owner of its value; the value will be deleted when the object ceases to exist.
- A `RefPPMonoidElem` is not the owner of its value, but the value may be changed (and the owner of the value will see the change too).
- A `ConstRefPPMonoidElem` is not the owner of its value, and its value may not be changed (through this reference).

The data layout is determined in `ConstRefPPMonoidElem`, and the more permissive classes inherit the data members. I have deliberately used a non-constant `PPMonoidElemRawPtr` for the value pointer as it is easier for the class `ConstRefPPMonoidElem` to add in constness appropriately than it is for the other two classes to remove it. The four assignment operators must all be defined since C++ does not allow polymorphism in the destination object (e.g. because of potential problems with slicing). Ideally it would be enough to define assignment just from a `ConstRefPPMonoidElem`, but I have to define also the "homogeneous" assignment operator since the default definition would not work properly. It is a bit tedious to have four copies of the relevant code (but it is only a handful of lines each time).

By convention the member functions of `PPMonoidBase` which operate on raw PP values assume that the values are valid (e.g. belong to the same PP monoid, division is exact in `myDiv`). The validity of the arguments is checked by the syntactically nice equivalent operations (see the code in PPMonoid.C). This permits a programmer to choose between safe clean code (with nice syntax) or faster unsafe code (albeit with uglier syntax).

58.4 Bugs, Shortcomings and other ideas

The section on "Advanced Use" is a bit out of date and too long.

- (1) Should more operations on `PPMonoidElem`s be inlined? With the current design, since speed is not so important for `PPMonoidElem`s.
- (2) We would like a way of performing divisibility tests faster when there are few indeterminates and relatively high degrees. In this case the DivMask is useless. The "gonnet" example is slow because it entails many divisibility tests. One suggestion would be to maintain a "randomly weighted" degree and use that as a simple heuristic for deciding quickly some cases.
- (3) I've fixed the various arithmetic functions for `PPMonoidElem`s so that they are obviously exception safe, BUT they now make an extra copy of the computed value (as it is returned from a local variable to the caller). Here is an idea for avoiding that extra copy. Create a new type (say PPMonoidElem_local) which offers just raw(..) and a function export(..) which allows the return mechanism to create a full `PPMonoidElem` (just by copying pointers) and empty out the PPMonoidElem_local. If the PPMonoidElem_local is not empty then it can destroy the value held within it. By not attempting to make PPMonoidElem_locals behave like full PPMonoidElems I save a lot of "useless" function definitions. Indeed the "export" function need not exist: an implicit ctor for a PPMonoidElem from a PPMonoidElem_local could do all the work. I'll wait to see profiling information before considering implementing.
- (4) Is assignment for `PPMonoid`s likely to be useful to anyone? I prefer to forbid it, as I suspect a program needing to use it is really suffering from poor design...
- (5) I have chosen not to use `operator^` for computing powers because of a significant risk of misunderstanding between programmer and compiler. The syntax/grammar of C++ cannot be changed, and `operator^` binds less tightly than (binary) `operator*`, so any expression of the form `a*b^c` will be parsed as `(a*b)^c`; this is almost certainly not what the programmer intended. To avoid such problems of misunderstanding I have preferred not to define `operator^`; it seems too dangerous.

- (6) The absence of a `deg` function for `PPMonoidElem`s is deliberate; you should choose either `StdDeg` or `wdeg` according to the type of degree you want to compute. This is unnatural; is is a bug?
- (7) I have deliberately not made the destructors for `ConstRefPPMonoidElem` and its descendants virtual. This is marginally risky: it might be possible to leak memory if you convert a raw pointer to `PPMonoidElem` into a raw pointer to `ConstRefPPMonoidElem`; of course, if you do this you're asking for trouble anyway.
- (8) Should `exponents` give an error if the values exceed the limits for `long`?
- (9) Offer the user some means of checking for and handling exponent overflow.

59 PPMonoidHom (John Abbott)

59.1 User documentation for the class PPMonoidHom

The class `PPMonoidHom` is used for representing homomorphisms between `PPMonoid`s. Each indeterminate in the domain monoid maps into an element of the codomain (*i.e.* a power product).

59.1.1 Examples

- ex-PPMonoidHom.C

59.1.2 Functions for PPMonoidHoms

Here is a list of the (pseudo-)ctors for `PPMonoidHom`

- `IdentityHom(PPM)` the identity
- `GeneralHom(PPM, images)` where `images` is a `vector` of `PPMonoidElem` whose i-th entry is the image of the i-th indet in `PPM`
- `RestrictionHom(PPM, IndetIndexes)` where `IndetIndexes` is a vector of indices of the indets which map to themselves, the others map to 1.

The `PPMonoidHom` object may be applied to a value by using normal function call syntax: for instance

```
PPMonoidElem t =  ...;
PPMonoidHom phi = ...;
cout << "phi applied to t gives " << phi(t) << endl;
```

Given a `PPMonoidHom` you can find out its domain and codomain:

`domain(phi)`	the domain of `phi` as a `PPMonoid`
`codomain(phi)`	the codomain of `phi` as a `PPMonoid`

59.2 Library Contributor Documentation

59.3 Maintainer documentation for PPMonoid, PPMonoidElem, and PPMonoidBase

59.4 Bugs, Shortcomings and other ideas

Add some more special cases: *e.g.* permutations of the indets, and the "identity" between PPMonoids which differ only in their orderings.

Should we allow partial homs? *e.g.* one which maps `x^2` to `y` (so odd powers of `x` have no image).

60 PPOrdering (John Abbott)

60.1 Examples

- ex-PPMonoidElem2.C
- ex-OrderingGrading1.C

60.2 User documentation

An object of the class `PPOrdering` represents an *arithmetic* ordering on the (multiplicative) monoid of power products, *i.e.* such that the ordering respects the monoid operation (viz. s < t => r*s < r*t for all r,s,t in the monoid).

In CoCoALib orderings and gradings are intimately linked (for gradings see also `degree` (Sec.17)).

60.2.1 Pseudo-constructors

Currently, the most typical use for a `PPOrdering` object is as an argument to a constructor of a concrete `PPMonoid` (Sec.58) or `PolyRing` (Sec.63), so see below **Convenience constructors**.

These are the functions which create new `PPOrdering`s:

- `NewLexOrdering(NumIndets)` – GradingDim = 0
- `NewStdDegLexOrdering(NumIndets)` – GradingDim = 1
- `NewStdDegRevLexOrdering(NumIndets)` – GradingDim = 1
- `NewMatrixOrdering(NumIndets, GradingDim, OrderMatrix)`

The first three create respectively lex, DegLex and DevRevLex orderings on the given number of indeterminates. Note the use of `Std` in the names to emphasise that they are only for standard graded polynomial rings.

The last function creates a `PPOrdering` given a matrix. `GradingDim` specifies how many of the rows of `OrderMatrix` are to be taken as specifying the grading.

Convenience constructors

For convenience there is also the class `PPOrderingCtor` which provides a handy interface for creating `PPMonoid` (Sec.58) and `SparsePolyRing` (Sec.87), so that `lex`, `StdDegLex`, `StdDegRevLex` may be used as shortcuts instead of the proper constructors, e.g.

```
NewPolyRing(RingQQ(), symbols("a","b","c","d"), lex);
```

is the same as

```
NewPolyRing(RingQQ(), symbols("a","b","c","d"), NewLexOrdering(4));
```

60.2.2 Queries

- `IsLex(PPO)` – true iff `PPO` is implemented as lex
- `IsStdDegLex(PPO)` – true iff `PPO` is implemented as StdDegLex
- `IsStdDegRevLex(PPO)` – true iff `PPO` is implemented as StdDegRevLex

60.2.3 Operations

The operations on a `PPOrdering` object are:

- `out << PPO` – output the `PPO` object to channel out
- `NumIndets(PPO)` – number of indeterminates the ordering is intended for
- `GradingDim(PPO)` – the dimension of the grading associated to the ordering
- `GetMatrix(PPO)` – a matrix defining the ordering

Thus CoCoALib supports graded polynomial rings with the restriction that the grading be compatible with the PP ordering: *i.e.* the grading comprises simply the first `k` entries of the *order vector*. The `GradingDim` is merely the integer `k` (which may be zero if there is no grading).

A normal CoCoA library user need know no more than this about `PPOrderings`. CoCoA Library contributors and the curious should read on.

There is also a member function (`M` a matrix) ... Don't use it yet!

- `PPO.myMatrixCopy(M)` – fill `M` with a matrix which specifies the ordering `PPO`

60.3 Maintainer documentation for PPOrdering

The general ideas behind the implementations of `PPOrdering` and `PPOrderingBase` are analogous to those used for `ring` and `RingBase`. `PPOrdering` is a simple reference counting smart-pointer class, while `PPOrderingBase` hosts the intrusive reference count (so that every concrete derived class will inherit it).

The only remaining observation to make about the simple class `PPOrdering` is that I have chosen to disable assignment – I find it hard to imagine when it could be useful to be able to assign `PPOrderings`, and suspect that allowing assignment is more likely to lead to confusion and poor programming style.

There are four concrete `PPOrderings` in the namespace `CoCoA::PPO`. The implementations are all simple and straightforward except for the matrix ordering which is a little longer and messier but still easy enough to follow.

60.4 Bugs, shortcomings and other ideas

Must resolve the restrictions on order matrices in `NewMatrixOrdering` (matrix must be square).

We need better ways to compose `PPOrderings`, *i.e.* to build new ones starting from existing ones. Max knows the sorts of operation needed here. Something similar to CoCoA4's `BlockMatrix` command is needed.

61 PPVector (Anna Bigatti)

61.1 class PPVector

WARNING THIS IS STILL A PRELIMINARY IMPLEMENTATION as indicated by the names of the implementation files: `TmpPPVector.H` and `TmpPPVector.C`.

This class is for dealing with lists of power-products.

This class has been designed to be used by monomial ideal operations, Hilbert, and Mayer-Vietoris trees.

The key functions interpret the list as generators of a monomial ideal: interreduction, divisibility test (ideal membership), lcms (ideal intersection).

The elements are `PPWithMask` (Sec.62), to make fast divisibility tests. Constructor requires a `PPMonoid` (Sec.58) and a `DivMaskRule` (see `DivMask` (Sec.23)), so that the user can choose the best suited implementations (mostly depending on the number of indeterminates and on the size of the exponents).

61.1.1 Examples

- ex-PPVector.C

61.2 Fields and main functions

Member fields are

```
PPMonoid myPPM;
DivMaskRule myDMR;
std::vector<PPWithMask> myVec;
```

61.2.1 Utility functions

- `PPMonoid PPM(const PPVector& PPs)`
- `DivMaskRule DMR(const PPVector& PPs)`
- `std::ostream& operator<<(std::ostream&, PPVector)`
- `bool IsEmpty(const PPVector& PPs)`
- `long len(const PPVector& PPs)`
- `void convert(std::vector<RingElem>& v, ring P, const PPVector& PPs)` converts PP's into RingElem's
- `void convert(PPVector PPs, const std::vector<RingElem>& v)` converts vector<RingElem> (if monomial!) into PPVector
- `void PushBack(PPVector& PPs, ConstRefPPMonoidElem pp)` if owner(pp) != PPM(PPs) it maps it
- `void PushBackPopBack(PPVector& ToPPs, PPVector& FromPPs)` move last PP from `FromPPs` into `ToPPs`
- `void swap(PPVector& PPs1, PPVector& PPs2)` swap PPs1 and PPs1

61.2.2 Mathemetical functions

- `bool IsDivisible(const PPWithMask& pp, const PPVector& ByL);` true is pp is divisible by an element of L
- `bool IsDivisible(ConstRefPPMonoidElem pp, const PPVector& ByL);` true is pp is divisible by an element of L
- `void interreduce(PPVector& PPs);` interreduce PPs (NOT exception clean)
- `void InterreduceSort(PPVector& PPs);` interreduce and sort PPs (NOT exception clean)
- `void lcms(PPVector& PPs, const PPVector& PPs1, const PPVector& PPs2);` all the lcm between elements of PPs1 and PPs2, effectively the generators of the intersection ideal

61.3 Bugs, Shortcomings and other ideas

61.3.1 Abstract Class

There was an attempt to make it an abstract class, `PPVectorBase`, made of abstract `PPVectorElem`, with the plan to have concrete classes a vector of `PPWithMask` (Sec.62), of `PPMonoidElem`, and of square-free pps (which cannot make a `PPMonoid` (Sec.58) because x*x is not square-free).

But this failed because most operations would need to know the type of the elements in the vector making it more suitable for templates. But

- (1) I much prefer inheritance (mathematically cleaner) and
- (2) probably all this generality is useless.

So it was sadly abandoned.

62 PPWithMask (Anna Bigatti)

62.1 Examples

- ex-PPWithMask1.C
- ex-PPWithMask2.C

62.2 User documentation

WARNING: THIS IS STILL ONLY A PRELIMINARY INTERFACE

A value of type `PPWithMask` is an "enriched" power product: it also contains a "divmask" so that divisibility tests between `PPWithMask` values can be effected quickly (on average, and assuming they are only rarely divisible).

To create a `PPWithMask` value:

```
PPWithMask(pp, DMRule)  -- pp is a power product, DMRule is a divmask rule
```

Given a `PPWithMask` value you can extract the power product and DivMaskRule using the following accessor functions:

```
PP(ppwm)          -- get a reference to the internal power product
DivMaskImpl(ppwm) -- get the div mask rule
```

Example:

```
DivMaskRule DMR = NewDivMaskEvenPowers();
PPMonoidElem PP1 ...
PPMonoidElem PP2 ...

PPWithMask pm1(PP1, DMR);
PPWithMask pm2(PP1, DMR);

IsEqualFast(pm1, pm2);
IsDivisibleFast(pm1, pm2);
```

Implementation of PPMonoidElem with DivMask for fast divisibility test

This type is not intended for "public use": it must be fast, so we cannot guarantee safety checks. It does some compatibility tests with CoCoA_ASSERT (i.e. only with `CoCoA_DEBUG` on)

It is to be used ONLY when speed on divisibility tests is crucial (Buchberger, Toric, Hilbert, ...).

Constructor and assignment from pp might be expensive.

62.3 Maintainer documentation for files BuildInfo

62.4 Bugs, Shortcomings and other ideas

Both impl and doc are very incomplete!

63 PolyRing (John Abbott)

63.1 User documentation for PolyRing

PolyRing is an abstract class (inheriting from `ring` (Sec.69)) representing rings of polynomials with coefficients in a commutative `ring` (Sec.69) `R`.

The polynomials may be (dense) univariate or (sparse) multivariate.

See `RingElem` PolyRing (Sec.71) for operations on its elements, but only a few operations are available at this level of abstraction: see `RingElem` SparsePolyRing (Sec.71) or `RingElem` DenseUPolyRing (Sec.71) for more operations on polynomials of `SparsePolyRing` (Sec.87) or `DenseUPolyRing` (Sec.20).

63.1.1 Examples

- ex-PolyRing1.C
- ex-PolyRing2.C
- ex-PolyIterator1.C
- ex-PolyIterator2.C
- ex-PolyInput1.C
- ex-factor1.C

63.1.2 Pseudo-constructors

Currently there are several functions to create polynomial rings: see `SparsePolyRing` constructors (Sec.87) for the sparse implementation and `DenseUPolyRing` constructors (Sec.20) for the dense implementation (currently only for univariate polynomials).

- `CoeffRing` is the ring of coefficients (must be commutative), - `NumIndets` specifies how many indeterminates there are; by default the indet names will be x[0],..x[NumIndets-1], and the ordering is `StdDegRevLex` – see `PPOrdering` (Sec.60).

- If the third parameter is specified then it is used in place of x in the indet names; we advise you to restrict to names comprising only letters (to be sure of future compatibility).

63.1.3 Queries and views

Let `R` be an object of type `ring` (Sec.69).

- `IsPolyRing(R)` – `true` if `R` is actually `PolyRing`
- `AsPolyRing(R)` – if `R` is a `PolyRing` *view* it as such

63.1.4 Operations on a PolyRing

In addition to the standard `ring` operations (Sec.69), a `PolyRing` (Sec.63) may be used in other functions.

Let `P` be an object of type `PolyRing`. Let `R` be an object of type `ring` (Sec.69).

- `NumIndets(P)` – the number of indeterminates in `P`
- `CoeffRing(P)` – the ring of coefficients of `P`
- `indets(P)` – a const std::vector of `RingElem` (Sec.71)s whose i-th element is the i-th indeterminate in `P`
- `indets(P, str)` – a std::vector of `RingElem` (Sec.71)s with all indeterminates in `P` whose head is the string `str`
- `indet(P,i)` – the `i`-th indet of `P` as a `RingElem` (Sec.71)
- `IndetPower(P,i,n)` – the `n`-th power of the `i`-th indet of `P` as a `RingElem` (Sec.71)

63.1.5 Homomorphisms

Let `P` be an object of type `PolyRing`. Let `R` be an object of type `ring` (Sec.69).

`CoeffEmbeddingHom(P)` the homomorphism which maps `CoeffRing(P)` into `P`

`PolyRingHom(P, R, CoeffHom, IndetImages)` the homomorphism from `P` to `R` which maps the coeffs using `CoeffHom`, and maps the k-th indet into `IndetImages[k]`

`EvalHom(P, IndetImages)` the evaluation homomorphism from `P` to `CoeffRing(P)` which is the identity on the coeffs, and maps the kth indet into `IndetImages[k]`

`PolyAlgebraHom(P, R, IndetImages)` must have `CoeffRing(P)` = `R` or `CoeffRing(P)` = `CoeffRing(R)` this is the identity on coeffs, and maps the k-th indet into `IndetImages[k]`

63.2 Maintainer documentation for PolyRing

The hard part has been deciding which member functions should be in `PolyRingBase` and which should be in less abstract classes. If you want to modify the code here, you should probably also look at `SparsePolyRing` and `DUPolyRing` before messing with the code!

The implementations in `PolyRing.C` are all very simple: they just conduct some sanity checks on the function arguments before passing them to the `PolyRing` member function which will actually do the work.

63.3 Bugs, Shortcomings and other ideas

What precisely should the *fancy* version of deriv do? What are permitted values for the second arg? Must coeff=1? What if the second arg does not have precisely one term?

The range of member functions on RawValues is rather a hotch-potch. Hopefully, experience and use of the code will bring some better order to the chaos.

Verify the true need for `myRemoveBigContent`, `myMulByCoeff`, `myDivByCoeff`. If the coeff ring has zero divisors then `myMulByCoeff` could change the structure of the poly!

Maintainer doc is largely absent.

64 QBGenerator (John Abbott)

64.1 User documentation for QBGenerator

The name `QBGenerator` derives from its intended use as a (monomial) *quotient basis generator*, that is a way of generating a factor closed (vector space) basis of power products for the quotient of a polynomial ring by a zero-dimensional ideal. It is used in the implementation of the **FGLM** and the **Buchberger-Moeller algorithms** – in fact these are really the same algorithm (for computing a Groebner basis of an intersection of one or more zero-dimensional ideals).

Background theory

Let `P` denote a polynomial `ring` (Sec.69) (with coefficients in a field `k`), and let `I` be a zero-dimensional `ideal` (Sec.39) in `P`. Then mathematically the quotient `P/I` is a finite dimensional vector space over `k`. We seek a basis `QB` for `P/I` which is a **factor closed** set of power products; *i.e.* if the power product `t` is in `QB` then any factor of `t` is in `QB` too. Groebner basis theory guarantees that such bases exist; actually it was first proved by Macaulay (a person, not a computer algebra system).

The elements of `QB` are determined one at a time, obviously starting with the trivial power product, 1. Moreover, at every stage the set of elements in the partially formed `QB` is factor closed, and this implies that only certain PPs are candidates for being adjoined to the `QB` (we call these **corners**). When a new element is adjoined to the `QB` new elements may appear in the *corner set*, these newly adjoined elements form the **new corner set** (this is always a subset of the *corner set*, and may be empty).

During the determination of the `QB`, some power products will be discovered which cannot be in the `QB` (usually based on the failure of a linear independence criterion). Such PPs form the **avoid set**: the `QBGenerator` will exclude all multiples of all elements of the *avoid set* from subsequent consideration.

64.1.1 Constructors and Pseudo-constructors

- `QBGenerator(PPM)` where `PPM` is the `PPMonoid` (Sec.58) in which we shall calculate; initially the *quotient basis* is empty, and the *corner set* contains just `1`.

64.1.2 Operations on QBGenerator

There are 3 accessor functions, and 2 true operations:

- `QBG.myQB()` gives the current elements of the *quotient basis* (as a `vector`) in the order they were added;
- `QBG.myCorners()` gives the current elements of the *corner set* (as a `list`);

- `QBG.myNewCorners()` gives the newly added elements to the *corner set* (as a `list`);
- `QBG.myCornerPPIntoQB(pp)` move the element `pp` of the *corner set* into the *quotient basis* (this updates both the *corner set* and the *new corner set*);
- `QBG.myCornerIntoAvoidSet(pp)` move the element `pp` of the *corner set* into the *avoid set* (all multiples of `pp` will skipped hereafter).

64.2 Maintainer documentation for QBGenerator

The tricky part was designing a good interface. The implementations themselves are relatvely straightforward (and actually contain some useful comments!)

The function `QBGenerator::myCornerPPIntoQB` makes local copies of some fields to permit full exception safety. This may adversely affect execution speed, but I believe that in the context of FGLM & Buchberger-Moeller the slow-down will be negligible *(but I have not actually tested my guess).*

64.3 Bugs, Shortcomings and other ideas

Class `QBGenerator` could offer a ctor which accepts a (good) estimate of the dimension of the quotient, *i.e.* final number of elements in the QB. It could use this value to `reserve` space for `myQBList`.

65 QuotientRing (John Abbott, Anna M. Bigatti)

65.1 User documentation for QuotientRing

A `QuotientRing` is an abstract class (inheriting from `ring` (Sec.69)) representing quotients of a `ring` (Sec.69) by an `ideal` (Sec.39).

See `RingElem` QuotientRing (Sec.71) for operations on its elements.

65.1.1 Examples

- ex-RingFp1.C
- ex-RingFp2.C

Extended example of use:

```
RingZZ ZZ = RingZZ();           // a copy of the ring of integers
ring ZZmod10 = NewZZmod(10);    // represents ZZ/(10) integers modulo 10
ring ZZmod10a = NewQuotientRing(ZZ, ideal(ZZ, 10)); // same as ZZmod10

RingHom phi = QuotientingHom(ZZmod10); // ring hom from ZZ to ZZmod10
RingElem r(ZZmod10, -3);         // an element of ZZmod10
RingElem preimage = CanonRepr(r);  // an element of ZZ = BaseRing(ZZmod10)

ring S = NewZZmod(2);            // another ring S, details do not matter much
RingHom theta = QuotientingHom(S); // any ring hom from ZZ to S will do instead
RingHom theta_bar = NewInducedHom(ZZmod10, theta); // induced ring hom from ZZmod10 to S
```

65.1.2 Constructors and Pseudo-constructors

- `NewQuotientRing(R, I)` – creates a new `ring` (Sec.69) representing the quotient `R/I`. `I` must be an ideal of `R`; odd things may happen if `I=R`. If `I` is zero then the result is isomorphic to `R` but not equal to `R`; arithmetic in `R` is more efficient than arithmetic in `R`/ideal(0).
- `NewZZmod(n)` – creates a new `ring` (Sec.69) representing the quotient `ZZ`/ideal(`n`) where `ZZ` is the ring of integers `RingZZ` (Sec.79) and `n` is an integer. A CoCoALib error will be thrown if `n`=1 or `n`=-1. Currently an error will be thrown also if `n`=0 (see BUGS).

NewZZmod or NewRingFp?

If `n` is a small prime then `NewZZmod(n)` produces the same result as `NewRingFp(n)` (or perhaps `NewRingFpDouble(n)`). If `n` is not a small prime then `NewRingFp(n)` throws an exception whereas `NewZZmod(n)` will produce a working quotient ring.

65.1.3 Query and cast

- `IsQuotientRing(R)` returns true iff R is implemented as a `QuotientRing`
- `AsQuotientRing(R)` returns a `QuotientRing` identical to the ring R if R is implemented as a `QuotientRing`, otherwise throws `ERR::NotQuotientRing`.

65.1.4 Operations on QuotientRing

In addition to the standard `ring` operations (Sec.69), a `QuotientRing` may be used in other functions.

Given `RmodI` a `QuotientRing` (representing R/I) built as `NewQuotientRing(R,I)` with `I` is an `ideal` (Sec.39) of the `ring` (Sec.69) `R`.

- `BaseRing(RmodI)` – a reference to the base ring of `RmodI` i.e. `R`
- `DefiningIdeal(RmodI)` – a reference to the ideal used to create `RmodI` i.e. `I`

65.1.5 Homomorphisms

- `QuotientingHom(RmodI)` – a reference to the quotienting homomorphism from `R` to `RmodI`
- `NewInducedHom(RmodI, phi)` – where `phi` is a `RingHom` (Sec.75) from `R` to `S`, creates a new `RingHom` (Sec.75) from `RmodI` to `S`. Two types of error may occur:
  - `ERR::BadInducingHom` – if `domain(phi)` is not `BaseRing(RmodI)`
  - `ERR::BadInducingHomKer` – if `phi` does not map the gens of `I` to zero.

65.2 Maintainer documentation for QuotientRing, QuotientRingBase, GeneralQuotientRingImpl

While considering the design of this code it may help to keep in mind these two canonical implementations:

`GeneralQuotientRingImpl` internally elements are represented as elements of a "representation ring" (which may differ from the base ring) which are kept reduced modulo some ideal (which may differ from the defining ideal)

`RingFpImpl` internally elements are represented by machine integers (see doc for RingFpImpl), a representation incompatible with that used for elements of the ring of integers (which is probably the base ring)

`QuotientRingBase` is an abstract class derived from RingBase, and is the base class for all quotient rings. It adds the following four new pure virtual member functions which must be defined in every concrete quotient ring:

```
virtual RingElem myCanonRepr(ConstRawValue r) const;
virtual void myReduction(RawValue& image, ConstRawValue arg) const;
virtual const RingHom& myQuotientingHom() const;
virtual RingHom myInducedHomCtor(const RingHom& InducingHom) const;
```

The member function myCanonRepr has to return a copy of the value since we cannot be sure that the internal representation is compatible with the internal representation of elements of the base ring.

65.3 Bugs, Shortcomings and other ideas

`IamTrueGCDDomain` always returns false. We can be clever in some easy cases, but it is hard in general (think of rings of algebraic integers which are gcd domains, but not euclidean domains).

Should NewZZmod(n) allow the case n==0? There's no mathematical reason to forbid it, but forbidding it may help detect programmer errors more quickly – it seems unlikely that one would really want to quotient by ideal(0).

FAIRLY SERIOUS CONFUSION: the code seems to make REPEATED sanity checks see

```
QuotientRingBase::QuotientRingBase
NewQuotientRing

QuotientRingHomBase::QuotientRingHomBase
NewInducedHom
```

I suspect that the C++ ctors should use CoCoA_ASSERT instead of checking always (and throwing an exception).

FURTHER SERIOUS CONFUSION: there is ambiguity about the difference between myBaseRing and myReprRing esp. for creating induced homomorphisms: given ring R, and ring S = R/I, create ring T = S/J An induced hom from T should start from a hom with domain S; or is it reasonable to accept a hom with domain R? In this case for T myReprRing is R but myBaseRing is S.

Given a RingHom from a QuotientRing it is not generally possible to obtain a reference to an "inducing hom": consider the hom from ZZ/(2) to ZZ/(2)[x] created by CoeffEmbeddingHom. A RingHom equivalent to the inducing hom can be produced by composing QuotientingHom with the given hom.

20 March 2004: I have added the possibility of making a trivial ring by quotienting: previously this was disallowed for no good reason that I could discern.

66 RandomSource (code: John Abbott; doc: John Abbott, Anna M. Bigatti)

66.1 Examples

- ex-RandomSource1.C
- ex-RandomSource2.C
- ex-RandomLong1.C
- ex-RandomBool1.C
- ex-RandomBigInt1.C

Here is a typical example of how to use a `RandomSeqLong`; note that we create the generator **before** entering the loop, then **inside** the loop we use the function `NextValue` to get 100 successive random values (between -9 and 9) from the generator:

```
RandomSeqLong rnd(-9,9);
for (int i=0; i < 100; ++i)
  cout << NextValue(rnd) << endl;
```

66.2 User documentation

Below, in `random` RandomSourceOperations (Sec.**??**) we list these handy functions for generating random values:

`RandomBool()`, `RandomLong(lo, hi)`, `RandomBigInt(lo, hi)`

they are probably just what you want in a simple program, **but using them will make your code thread-unsafe** (which is quite acceptable in a small program for personal use).

For a **thread-safe** solution you should create your own *random generator*. If you just want to generate many random values of the same type, you should consider using one of the three specialized random sequence generators (which are faster than the more general `RandomSource`):

- The class `RandomSeqLong` is for representing generators of (independent) uniformly distributed random integers (`long`) in a given range; the range is specified when creating the generator (and cannot later be changed).
- The class `RandomSeqBigInt` is for representing generators of (independent) uniformly distributed random integers (`BigInt` (Sec.8)) in a given range; the range is specified when creating the generator (and cannot later be changed).
- The class `RandomSeqBool` models a binary random variable (with independent `bool` samples, each having 50% probability of being *true* and 50% of being *false*).
- The class `RandomSource` is for representing general sources of (pseudo-)randomness: you can use it to produce random `bool`, `long`, and `BigInt` values.

66.2.1 Constructors

All constructors have an optional argument which is the initial *seed* – it determines the initial state of the generator. If you do not give a seed, the default is `0`.

If you create several random sequence generators of the same kind and with the same seed, they will each produce exactly the **same sequence of values**. If you want to obtain different results each time a program is run, you can seed the generator with the system time (*e.g.* by supplying as argument `time(0)`); this is likely desirable unless you're trying to debug a randomized algorithm.

For `RandomSource` see also the `reseed` function documented below in *RandomSource Operations*.

RandomSource

`RandomSource()`	seeded with `0`
`RandomSource(n)`	seeded with `n`

For convenience, there is a global `RandomSource` object (belonging to `GlobalManager` (Sec.37)); you can get a reference to it by calling `GlobalRandomSource()`, **but using it will make your code thread-unsafe**.

RandomSeqXXXX

`RandomSeqBigInt(lo,hi)`	seeded with `0`
`RandomSeqLong(lo,hi)`	seeded with `0`
`RandomSeqBool()`	seeded with `0`
`RandomSeqBigInt(lo,hi, n)`	seeded with `abs(n)`
`RandomSeqLong(lo,hi, n)`	seeded with `abs(n)`
`RandomSeqBool(n)`	seeded with `abs(n)`

Each `RandomSeqBigInt` (or `RandomSeqLong`) will produce (pseudo) random values uniformly distributed in the range from `lo` to `hi` (with both extremes included). An `ERR::BadArg` exception is thrown if `lo` $>$ `hi`; the case `lo == hi` is allowed.

66.2.2 RandomSource Operations

These are the most convenient functions for generating random values; but they use `GlobalRandomSource`, so they are **thread-unsafe**:

- `RandomBool()` – returns `true` with probability 50%
- `RandomBiasedBool(P)` – returns `true` with probability `P`
- `RandomLong(lo, hi)` – in range `lo`..`hi` (both ends included)
- `RandomBigInt(lo, hi)` – in range `lo`..`hi` (both ends included)

A cleaner way is to pass as an argument the specific `RandomSource` object to be used (in these examples we call it `RndSrc`):

- `RandomBool(RndSrc)`
- `RandomLong(RndSrc, lo, hi)` – in range `lo..hi` (both ends included)
- `RandomBigInt(RndSrc, lo, hi)` – in range `lo..hi` (both ends included)

A `RandomSource` object may be reseeded at any time; immediately after reseeding it will generate the same random sequence as a newly created `RandomSource` initialized with that same seed. The seed must be an integer value.

- `reseed(RndSrc, seed)`

Note about thread-safety: the various operations on a fixed `RandomSource` object are not thread-safe; to achieve thread safety, you should use *different objects in different threads*. In particular, it is best not to use `GlobalRandomSource()` in a multi-threaded environment.

66.2.3 RandomSeqXXXX Operations

Once you have created a `RandomSeqXXXX` you may perform the following operations on it:

- `*rnd` – get the current value of rnd (as a boolean).
- `++rnd` – advance to next value of rnd.
- `rnd++` – advance to next value of rnd **INEFFICIENTLY**.
- `NextValue(rnd)` – advance and then return new value; same as *++rnd
- `out << rnd` – print some information about rnd.
- `rnd.myIndex()` – number of times rnd has been advanced, (same as the number of random bools generated).

  Note that a `RandomSeqXXXX` supports input iterator syntax.

  Moreover, for `RandomSeqBool` there is a special function
- `NextBiasedBool(RndBool, P)` – use several samples from `RndBool` to produce a boolean with probability `P` of being true; may consume many values from `RndBool` but on average consumes less than 2 values per call.

You may assign or create copies of `RandomSeqXXXX` objects; the copies acquire the complete state of the original, so will go on to produce exactly the same sequence of values as the original will produce.

66.3 Maintainer documentation

RandomSource

The impl is mostly quite straightforward since almost all the work is done by GMP.

`RandomLong(RndSrc, lo, hi)` is a bit messy for two reasons:

1. CoCoALib uses signed longs while GMP uses unsigned longs;
2. the case when (`lo,hi`) specify the whole range of representable longs requires special handling.

RandomSeqLong and RandomSeqBigInt

The idea is very simple: use the pseudo-random number generator of GMP to generate a random machine integer in the range 0 to `myRange-1` (where `myRange` was set in the ctor to be `1+myUpb-myLwb`) and then add that to `myLwb`. The result is stored in the data member `myValue` so that input iterator syntax can be supported.

There are two *non essential* data members: `mySeed` and `myCounter`. I put these in to help any poor blighter who has to debug a randomized algorithm, and who may want to *fast forward* the random sequence to the right place.

The data member `myState` holds all the state information used by the GMP generator. Its presence makes the ctors, dtor and assignment messier than they would have been otherwise.

The advancing and reading member functions (*i.e.* `operator++` and `operator*`) are inline for efficiency, as is the `NextValue` function.

`myGetValue` is a little messy because the value generated by the GMP function `gmp_urandomm_ui` cannot generate the full range of `unsigned long` values. Instead I have to call `gmp_urandomb_ui` if the full range is needed.

The data members `myLwb`, `myUpb` and `myRange` are morally constant, but I cannot make them `const` because I wanted to allow assignment of `RandomSeqLong` objects.

RandomSeqBool

The idea is very simple: use the pseudo-random number generator of GMP to generate a random integer, and then give out the bits of this integer one at at time; when the last bit has been given out, get a new random integer from the GMP generator. The random integer is kept in the data member `myBuffer`, and `myBitIndex` is used to read the bits one at a time.

The condition for refilling `myBuffer` is when the index goes beyond the number of bits held in `myBuffer`.

`myFillBuffer` also sets the data member `myBitIndex` to zero; it seemed most sensible to do this here.

The function `prob` is nifty; if you think about it for a moment, it is obviously right (and economical on random bits). It would be niftier still if the probability were specified as an `unsigned long` – on a 64-bit machine this ought to be sufficient for almost all purposes. If the requested probability is of the form N/2^k for some odd integer N, then the average number of bits consumed by `prob` is 2-2^(1-k), which always lies between 1 and 2. If the requested probability is 0 or 1, no bits are consumed.

66.4 Bugs, shortcomings and other ideas

The printing function gives only partial information; *e.g.* two `RandomSource` objects with different internal states might be printed out identically.

The implementation simply calls the GMP pseudo-random generator; this generator is deterministic (so always produces the same sequence), but if you change versions of GMP, the sequence of generated values may change. You will have to read the GMP documentation to know more.

Discarded idea: have a ctor which takes a ref to a `RandomSource` (Sec.66), and which uses that to obtain randomness. I discarded the idea because of the risks of an **invisible external reference** (*e.g.* a dangling reference, or problems in multithreaded code). Instead of passing a reference to a `RandomSource` (Sec.66) to the ctor, you can use the `RandomSource` (Sec.66) to create an initial seed which is handed to the ctor – this gives better separation.

Why can `RandomSource` be seeded with a `BigInt` but the others not? Does anyone really care?

66.4.1 Doubts common to RandomSeqBigInt, RandomSeqBool, RandomSeqLong

It might be neater to put `++myCounter` inside `myGenValue`, though this would mean that `myCounter` gets incremented inside the ctor.

Should `NextValue` advance **before** or **after** getting the value?

Is the information printed by `myOutputSelf` adequate? Time will tell.

Is there a better way of writing the four ctors (for `RandomSeqBigInt`) without repeating many lines of essentially identical source code?

Are there too many inline fns? Is run-time speed so important here? How many algorithms really consume

millions of random bits/numbers? Surely the computations on the random values should exceed the cost of generating them, shouldn't they?

66.5 Main changes

- **December 2012 (v0.9953)**
  - major rewriting: now all classes are defined in one single file `random.[HC]`
  - some classes and functions have been renamed: `RandomXXXXSource` to `RandomSeqXXXX`, and `sample` to `NextValue`
  - documentation is unified in `random.txt`
- **October 2012 (v0.9952)**
  - clarified doc; added comments about thread-safety.
- **February 2011 (v0.9949)**
  - removed `RandomLong(src)` (*i.e.* with no range)
  - added `RandomBool()`, `RandomLong(lo,hi)`, `RandomBigInt(lo,hi)` (*i.e.* with no `RandomSource`)
- **April 2011 (v0.9943)** first release

67 ReductionCog (Anna Bigatti)

67.1 class ReductionCogBase

ReductionCogBase is an abstract class to perform a full reduction:

it contains two parts: "IgnoredPPs" summands whose PPs are to be ignored "Active" the part which will be reduced Thanks to the limited operations allowed on a ReductionCog, all PP in IgnoredPPs are gueranteed bigger than the PPs in the Active part.

with a ReductionCog F you can compute:

ActiveLPP(F) the LPP of the Active part IsActiveZero(F) is the Active part zero?

F.myMoveToNextLM() move the LM of the Active part to the IgnoredPPs F.myReduce(f) reduce the Active part with f F.myAssignReset(f) the Active part gets f; f and IgnoredPPs get 0 F.myAssignReset(f, fLen) same as above but faster for geobucket implementation F.myRelease(f) F gets the total value of f; f gets 0 F.myOutput(out)

The idea is that LM will be reduced first, if the result is not 0 it will be "set aside and ignored" and the new LM of the Active part will be reduced, and so on.

The result of myReduce is defined up to a constant (in the coefficients ring)

Constructors are

```
ReductionCog NewRedCogPolyField(const SparsePolyRing& P);
ReductionCog NewRedCogPolyGCD(const SparsePolyRing& P);
ReductionCog NewRedCogGeobucketField(const SparsePolyRing& P);
ReductionCog NewRedCogGeobucketGCD(const SparsePolyRing& P);
```

If "GCD" myRelease makes poly content free, but if "Field: myRelease does NOT make poly monic. ... I can't remember why I made this choice....

example

```
ReductionCog F = ChooseReductionCogGeobucket(myGRingInfoValue);
F->myAssignReset(f, fLen);
while ( !IsActiveZero(F) )
{
  (..) // find reducer g or break
  F->myReduce(g);
}
F->myRelease(f);
```

67.2 implementations

in general the geobucket implementation are to be preferred

`RedCog::PolyFieldImpl` this implementation contains two RingElem.

`RedCog::PolyGCDImpl` this implementation contains two polynomials [RingElem] two coefficients [RingElem] and a counter

`RedCog::GeobucketFieldImpl` this implementation contains a RingElem for the IgnoredPPs and a geobucket for the Active part

`RedCog::GeobucketGCDImpl` this implementation contains a RingElem for the IgnoredPPs and a geobucket for the Active part two coefficients [RingElem] and a counter

68 RegisterServerOps (Anna Bigatti)

68.1 User documentation

68.1.1 Quick and easy way to add a single operation

When you want to have some operation accessible from CoCoA-4 you need to make these steps:

- integrate your operation into CoCoALib
  - make `TmpMyFile.[HC]`
  - add `TmpMyFile.C` to `src/AlgebraicCore/Makefile`
  - add `TmpMyFile.H` to `include/library.H`
- make a `ServerOpBase` for it in `RegisterServerOpsUser.C` (see `ServerOp` (Sec.80))
- register it in `RegisterServerOps.C` (see below)

Register your ServerOpBase in `bool RegisterOps()` at the end of the file:

```
void RegisterOp(const std::string& s, ServerOp o);
```

where `s` is the "OpenMath name" of the operation for the communication with CoCoA-4 (used in cocoa5.cpkg).

Properly, you need to choose 3 names for your operation:

- the **CoCoALib name** for the `ServerOp` (following the CoCoALib coding conventions)
- the **"OpenMath" name** used only for computer communication
- the **CoCoA-4 name** for the CoCoA-4 user (following the CoCoA-4 conventions and ending with a "5" to mean **CoCoA-5**)

68.1.2 Proper way to add a library

You should make a dedicated file `RegisterServerOpsMyOperations.C` (see, for example, `src/AlgebraicCore/RegisterServerOps`

Then you should choose a meaningful name for the namespace of your operations (for example `CoCoAServerOperationsFromFro`) and define your own `RegisterOps` and copy the function `RegisterOpsOnce`:

```
namespace NamespaceForMyOperations
{
  bool RegisterOps()
  {
    RegisterOp("OpenMathName1", ServerOp(new CoCoALibName1()));
    RegisterOp("OpenMathName2", ServerOp(new CoCoALibName2()));
    ...
    return true;
```

```
  }


  bool RegisterOpsOnce()
  {
    static bool EvalOnce = RegisterOps();
    return EvalOnce;
  }
}
```

Then add in `src/server/RegisterServerOps.H` the registration of your operations simply copying these lines:

```
namespace NamespaceForMyOperations
{
  bool RegisterOpsOnce();
  bool GlobalDummyVar = RegisterOpsOnce();
}
```

or make a dedicated file `MyRegisterServerOps.H` (see, for example, `src/server/RegisterServerOpsFrobby.H`) and include it in `src/server/CoCoAServer.C`

68.2 Mantainer documentation

How does this work? When `CoCoAServer.C` is compiled the global variables are initialized.

Therefore `NamespaceForMyOperations::GlobalDummyVar` which is declared in the included file `RegisterServerOps.H` is initialized by calling `NamespaceForMyOperations::RegisterOpsOnce()` with the *side effect* of registering your operations.

68.3 Main changes

68.3.1 2009

Cleaned up the documentation after integration of the Frobby library.

69 ring (John Abbott, Anna M. Bigatti)

69.1 User documentation

The primary use for a variable of type `ring` is simply to specify the ring to which `RingElem` (Sec.71) variables are associated. CoCoALib requires that the user specify first the rings in which to compute, then values in those rings can be created and manipulated. We believe that this explicit approach avoids any possible problem of ambiguity.

The file `ring.H` introduces several classes used for representing rings and their elements. A normal user of the CoCoA library will use principally the classes `ring` and `RingElem` (Sec.71): an object of type `ring` represents a mathematical ring with unity, and objects of type `RingElem` (Sec.71) represent values from some ring. To make the documentation more manageable it has been split into two: this file describes operations directly applicable to rings, whereas a separate file describes the operations on a `RingElem` (Sec.71). Documentation about the creation and use of homomorphisms between rings can be found in `RingHom` (Sec.75).

The documentation here is very general in nature: it applies to all rings which can be created in the CoCoA library. To find out how to create rings, and for more specific documentation about the various special types of ring CoCoALib offers, look at the relevant file: see the subsection below about ***Types of Ring***.

While the CoCoA library was conceived primarily for computing with commutative rings, the possibility of creating and using certain non-commutative rings exists. The documentation for these rings is kept separately in `RingWeyl` (Sec.78).

69.1.1 Examples

Here is a list of example programs (to be found in the `examples/` subdirectory) illustrating the creation and use of various sorts of ring and their elements

- ex-ring1.C
- ex-ring2.C
- ex-RingElem1.C
- ex-RingFp1.C
- ex-RingFp2.C
- ex-RingQ1.C
- ex-RingTwinFloat1.C
- ex-RingWeyl1.C
- ex-RingZZ1.C

69.1.2 Types of ring (inheritance structure)

- `RingZZ` (Sec.79)
- `RingTwinFloat` (Sec.77)
- `FractionField` (Sec.32)
  - generic
  - `RingQQ` (Sec.76)
- `QuotientRing` (Sec.65)
  - generic
  - `RingFp` (Sec.72)
  - `RingFpLog` (Sec.74)
  - `RingFpDouble` (Sec.73)
  - Simple algebraic extensions (not yet implemented)
- `PolyRing` (Sec.63)
  - `DenseUPolyRing` (Sec.20)
    - `DenseUPolyClean` (Sec.19)
  - `SparsePolyRing` (Sec.87)
    - `RingWeyl` (Sec.78)
    - `DistrMPoly` (Sec.21)
    - DistrMPolyInlPP
    - `DistrMPolyInlFpPP` (Sec.??)

69.1.3 Pseudo-constructors

The default initial value for a `ring` is the ring of integers (`RingZZ`).

You can specify explicitly the initial value using one of the various ring pseudo-constructors:

`RingZZ()`	see `RingZZ` constructors (Sec.79)
`RingQQ()`	see `RingQQ` constructors (Sec.76)
`NewZZMod(n)`	see `QuotientRing` constructors (Sec.65)
`NewRingTwinFloat(n)`	see `RingTwinFloat` constructors (Sec.77)
`NewFractionField(R)`	see `FractionField` constructors (Sec.32)
`NewQuotientRing(R,I)`	see `QuotientRing` constructors (Sec.65)

69.1.4 Operations on Rings

Let `R` and `R2` be two variable of type `ring`.

- `characteristic(R)` – the characteristic of `R` (as a `BigInt` (Sec.8))
- `symbols(R)` – a std::vector of the symbols in `R` (*e.g.* Q(a)[x,y] contains the symbols a, x, and y)
- `R = R2` – assign `R2` to `R` (so they both refer to the same identical internal impl)
- `R == R2` – test whether `R` and `R2` are **identical** (*i.e.* they refer to the same internal impl)
- `R != R2` – the logical negation of `R == R2`

`zero(R)`	the zero element of `R`
`one(R)`	the one element of `R`

Queries

In some cases the best algorithm to use may depend on whether the ring in which we are computing has certain properties or not; so CoCoALib offers some functions to ask a ring `R` about its properties:

- `IsCommutative(R)` – a boolean, true iff `R` is commutative
- `IsIntegralDomain(R)` – a boolean, true iff `R` has no zero divisors
- `IsTrueGCDDomain(R)` – a boolean, true iff `R` is a true GCD domain (**note**: fields are *not* true GCD domains)
- `IsOrderedDomain(R)` – a boolean, true iff `R` is arithmetically ordered
- `IsField(R)` – a boolean, true iff `R` is a field
- `IsFiniteField(R)` – a boolean, true iff `R` is a finite field
- `LogCardinality(R)` – the integer k such that card(R) = pˆk where p is char(R)

NOTE: a pragmatic approach is taken: *e.g.* `IsOrderedDomain` is true only if comparisons between elements can actually be performed using the CoCoA library.

Queries and views

It may also be important to discover practical structural details of a `ring` (*e.g.* some algorithms make sense only for a polynomial ring). The following query functions `Is...` tell you how the `ring` is implemented, and the *view* functions `As...` give access to the specific operations:

- `IsZZ(R)` – see `RingZZ` query (Sec.79)
- `IsQQ(R)` – see `RingQQ` query (Sec.76)
- `IsDenseUPolyRing(R)` – see `DenseUPolyRing` query (Sec.20)
- `IsFractionField(R)` – see `FractionField` query (Sec.32)
- `IsPolyRing(R)` – see `PolyRing` query (Sec.63)
- `IsQuotientRing(R)` – see `QuotientRing` query (Sec.65)
- `IsSparsePolyRing(R)` – see `SparsePolyRing` query (Sec.87)

In general the function "IsXYZ" should be read as "Is internally implemented as XYZ": for instance `IsQuotientRing` is true only if the internal implementation is as a quotient ring, so if `ZZ` denotes the ring of integers and `R = ZZ[x]/ideal(x)` then `R` and `ZZ` are obviously isomorphic but `IsZZ(R)` gives `false` and `IsZZ(Z)` gives `true`, while conversely `IsQuotientRing(R)` gives `true` and `IsQuotientRing(ZZ)` gives `false`.

69.1.5 ADVANCED USE OF RINGS

The rest of this section is for more advanced use of `ring`s (*e.g.* by CoCoA library contributors). If you are new to CoCoA, you need not read this subsection.

Writing C++ classes for new types of ring

An important convention of the CoCoA library is that the class `RingBase` is to be used as an abstract base class for all rings. You are strongly urged to familiarize yourself with the maintainer documentation if you want to understand how and why rings are implemented as they are in the CoCoA library.

The first decision to make when implementating a new ring class for CoCoALib is where to place it in the ring inheritance structure. This inheritance structure is a (currently) tree with all concrete classes at the leaves, and all abstract classes being internal nodes. Usually the new concrete ring class is attached to the structure by making it derive from one of the existing abstract ring classes. You may even decide that it is appropriate to add a new abstract ring class to this structure, and to make the new concrete class derive from this new abstract class.

Note: I have note used multiple inheritance in the structure, largely because I do not trust multiple inheritance (not doubt due in part to my ignorance of the topic).

Once you have decided where to attach the new concrete class to the structure, you will have to make sure that all pure virtual functions in the abstract class are implemented. Almost all instances of concrete rings are built through pseudo-constructors (the rings ZZ and QQ are exceptional cases).

An **important detail** of the constructor for a new concrete ring is that the reference count of the new ring object must be incremented to 1 at the start of the constructor body (or more precisely, before any self references are created, *e.g.* when creating the zero and one elements of the ring); without this "trick" the constructor is not exception safe.

NOTE Every concrete ring creates a copy of its zero and one elements (kept in auto_ptrs `myZeroPtr` and `myOnePtr`). This common implementation detail cannot (safely) be moved up into `RingBase` because during destruction by default the data members of `RingBase` are destroyed after the derived concrete ring. It seems much safer simply to duplicate the code for each ring implementation class.

69.2 Maintainer documentation

(**NB** consider consulting also `QuotientRing` (Sec.65), `FractionField` (Sec.32) and `PolyRing` (Sec.63))

The design underlying rings and their elements is more complex than I would have liked, but it is not as complex as the source code may make it appear. The guiding principles are that the implementation should be flexible and easy/pleasant to use while offering a good degree of safety; extreme speed of execution was not a goal (as it is usually contrary to good flexibility) though an interface offering slightly better run-time efficiency remains.

Regarding flexibility: in CoCoALib we want to handle polynomials whose coefficients reside in (almost) any commutative ring. Furthermore, the actual rings to be used will be decided at run-time, and cannot restricted to a given finite set. We have chosen to use C++ inheritance to achieve the implementation: the abstract class `RingBase` defines the interface that every concrete ring class must offer.

Regarding ease of use: since C++ allows the common arithmetic operators to be overloaded, it is essential that these work as expected for elements of arbitrary rings – with the caveat that `/` means **exact division**, as this is the only reasonable interpretation. Due to problems of ambiguity, arithmetic between elements of different rings is forbidden: *e.g.* let f be in QQ[x,y] and g in ZZ[y,x], where should f+g reside?

The classes in the file ring.H are closely interrelated, and there is no obvious starting point for describing them – you may find that you need to read the following more than once to comprehend it. Here is a list of the classes:

`ring`	value represents a ring; it is a smart pointer
`RingBase`	abstract class *defining what a ring is*
`RingElem`	value represents an element of a ring
`ConstRefRingElem`	const-reference to a RingElem
`RingElemConstRawPtr`	raw pointer to a *const* ring value
`RingElemRawPtr`	raw pointer to a ring value

The class `RingBase` is of interest primarily to those wanting to implement new types of ring (see relevant section below); otherwise you probably don't need to know about it. Note that `RingBase` includes an intrusive reference counter – so every concrete ring instance will have one. `RingBase` also includes a machine integer field containing a

unique numerical ID – this is so that distinct copies of otherwise identical rings can be distinguished when output (*e.g.* in OpenMath).

The class `ring` is used to represent mathematical rings (*e.g.* possible values include ZZ, QQ, or QQ[x,y,z]). An object of type `ring` is just a reference counting smart pointer to a concrete ring implementation object – so copying a ring is fairly cheap. In particular two rings are considered equal if and only if they refer to the same identical concrete ring implementation object. In other files you will find classes derived from `ring` which represent special subclasses of rings, for instance `PolyRing` is used to represent polynomial rings. The intrusive reference count, which must be present in every concrete ring implementation object, is defined as a data member of `RingBase`.

For the other classes see `RingElem` (Sec.71).

Further comments about implementation aspects of the above classes.

Recall that `ring` is essentially a smart pointer to a `RingBase` object, *i.e.* a concrete implementation of a ring. Access to the implementation is given via `operator->`. If necessary, the pointer may also be read using the member function `myRingPtr`: this is helpful for defining functions such as `IsPolyRing` where access to the pointer is required but `operator->` cannot be used.

The class `RingBase` declares a number of pure virtual functions for computing with ring elements. Since these functions are pure they must all be fully defined in any instantiable ring class (*e.g.* RingZZImpl or RingFpImpl). These member functions follow certain conventions:

RETURN VALUES: most arithmetic functions return no value, instead the result is placed in one of the arguments (normally the first argument is the one in which the result is placed), but functions which return particularly simple values (*e.g.* booleans or machine integers) do indeed return the values by the usual function return mechanism.

ARG TYPES: ring element values are passed as *raw pointers* (*i.e.* a wrapped `void*` pointing to the actual value). A read-only arg is of type `RingElemConstRawPtr`, while a writable arg is of type `RingElemRawPtr`. When there are writable args they normally appear first. For brevity there are typedefs `ConstRawPtr` and `RawPtr` in the scope of `RingBase` or any derived class.

ARG CHECKS: sanity checks on the arguments are **not conducted** (*e.g.* the division function assumes the divisor is non-zero). These member functions are supposed to be fast rather than safe. However, if the compilation flag `CoCoA_DEBUG` was set then some checks may be performed.

In a few cases there are non-pure virtual member functions in `RingBase`. They exist either because there is a simple universal definition or merely to avoid having to define inappropriate member functions (*e.g.* gcd functions when the ring cannot be a gcd domain). Here is a list of them:

`IamTrueGCDDomain()`	defaults to not `IamField()`
`IamOrderedDomain()`	defaults to false

69.3 Bugs, Shortcomings and other ideas

Printing rings is unsatisfactory. Need a mechanism for specifying a print name for a ring; and also a mechanism for printing out the full definition of the ring avoiding all/some print names. For instance, given the definitions `R = QQ(x)` and `S = R[a,b]` we see that `S` could be printed as `S`, `R[a,b]` or `QQ(x)[a,b]`. We should allow at least the first and the last of these possibilities.

Should (some of) the query functions return `bool3` values? What about properties which are hard to determine?

There used to be an `IsFinite(R)` function in the documentation, but it did not exist in the code. Should it exist?

70 RingDistrMPoly (John Abbott)

70.1 User documentation for the class RingDistrMPoly

`RingDistrMPoly` implements a ring of distributed multivariate polynomials: you may think of the elements as being ordered lists of coefficient and power product pairs (with the additional guarantee that the coefficients are

non-zero, and that the power products are all distinct. The best way to create a `RingDistrMPoly` is to use the function `NewPolyRing` (see `PolyRing` (Sec.63))

A `RingDistrMPoly` is a concrete instance of a `SparsePolyRing` (Sec.87).

70.2 Maintainer documentation for the class RingDistrMPoly

I have implemented `HomImpl` and `IdealImpl` as private subclasses since their existence should not be known outside the scope of the class `RingDistrMPolyImpl`.

70.2.1 Bugs and Shortcomings

Documentation almost completely absent.

The implementation of `RingDistrMPolyImpl::HomImpl::myApply` is very poor. I simply needed some easy code that would work. A major overhaul will be needed when I have understood how best to implement it.

71 RingElem (John Abbott)

71.1 Examples

- ex-RingElem1.C
- ex-RingFp1.C
- ex-RingFp2.C
- ex-RingQ1.C
- ex-RingTwinFloat1.C
- ex-RingWeyl1.C
- ex-RingZZ1.C
- ex-PolyRing3.C
- ex-NF.C

71.2 User documentation

The file `ring.H` introduces several classes used for representing rings and their elements. A normal user of the CoCoA library will use principally the classes `ring` (Sec.69) and `RingElem`: an object of type `ring` (Sec.69) represents a mathematical ring with unity, and objects of type `RingElem` represent values from some ring. To make the documentation more manageable it has been split into two: this file describes operations on a `RingElem`, whereas a separate file describes the operations directly applicable to `ring` (Sec.69)s. Documentation about the creation and use of homomorphisms between rings can be found in `RingHom` (Sec.75).

An object of type `RingElem` comprises two internal parts: the ring to which the value belongs, and the value itself. For instance, this means that the zero elements of different rings are quite different objects.

71.2.1 Constructors

Normally when creating a new `RingElem` we specify both the ring to which it belongs, and its initial value in that ring. Let `R` be a `ring` (Sec.69). Let `n` be a machine integer or a `BigInt` (Sec.8). Let `q` be a rational, *i.e.* a value of type `BigRat` (Sec.9). Let `r2` be a ring element.

`RingElem r(R);`	an element of `R`, initially 0
`RingElem r(R, n);`	an element of `R`, initially the image of `n`
`RingElem r(R, q);`	an element of `R`, initially the image of `q` (or error)
`RingElem r(R, s);`	an element of `R`, initially the value of `symbol` (Sec.90) `s`
`RingElem r(R, r2);`	an element of `R`, maps `r2` into `R` via `CanonicalHom`

Construction from a rational may fail, *e.g.* if the denominator is a zero divisor in the ring; if it does fail then an exception is thrown (with code `ERR::DivByZero`).

You can create a copy of a ring element in the usual way:

`RingElem r(r2);`	a copy of `r2`, element of the same ring
`RingElem r = r2;`	(alternative syntax, discouraged)

If no ring is specified when creating a `RingElem` then the new value belongs to `RingZZ()`:

`RingElem r;`	an element of `RingZZ()`, initially zero
`RingElem r(n);`	an element of `RingZZ()`, initially the image of `n`
`RingElem r(q);`	an element of `RingZZ()`, initially the image of `q`

Naturally the last constructor works only if the denominator of the rational `q` is 1.

These are not really constructors: you can get the zero and one of a `ring` (Sec.69) directly using the following:

`zero(R)`	the zero element of `R`
`one(R)`	the one element of `R`

71.2.2 Operations on RingElems

`RingElem`s are designed to be easy and safe to use; the various checks do incur a certain run-time overhead, so a faster alternative is offered (see below in the section *Fast and Ugly Code*). Arithmetic operations between `RingElem`s will fail if they do not belong to the same ring (the exception has code `ERR::MixedRings`).

Assignment & Swapping

Assigning an integer or rational to a `RingElem` wil automatically map the value into the ring to which the `RingElem` belongs.

`r = n;`	map `n` into `owner(r)` and assign the result
`r = q;`	map `q` into `owner(r)` and assign the result
`r = r2;`	`r` becomes a copy of `r2` – afterwards `owner(r)==owner(r2)`
`swap(r,s);`	exchange the values (and the owning rings)

Arithmetic

Arithmetic operations between `RingElem`s will fail if they do not belong to the same ring (the exception has code `ERR::MixedRings`). You may perform arithmetic between a `RingElem` and a machine integer, a `BigInt` value or a `BigRat` value – the integer/rational is automatically mapped into the same ring as the `RingElem`.

Let `r` be a non-const `RingElem`, and `r1`, `r2` be potentially const `RingElem`s. Assume they are all associated to the same ring. Then the operations available are: (meanings are obvious)

- `cout << r1` – output value of `r1` (decimal only, see notes)
- `r1 == r2` – equality test
- `r1 != r2` – not-equal test
- `-r1` – negation (unary minus)
- `r1 + r2` – sum
- `r1 - r2` – difference
- `r1 * r2` – product
- `r1 / r2` – quotient, division must be exact (see `IsDivisible`)
- `r += r1` – equivalent to `r = r + r1`

- `r -= r1` – equivalent to `r = r - r1`
- `r *= r1` – equivalent to `r = r * r1`
- `r /= r1` – equivalent to `r = r / r1`
- `power(r1, n)` – `n`-th power of `r1`; `n` any integer
- `r^n` – **THIS DOES NOT WORK!!!** it does not even compile, you must use `power`

Attempting to compute a `gcd` or `lcm` in a ring which not an effective GCD domain will produce an exception with code `ERR::NotTrueGCDDomain`. If `r1` or `r2` is a `BigRat` then an error is signalled at compile time.

- `gcd(r1, r2)` – an associate of the gcd
- `lcm(r1, r2)` – an associate of the lcm
- `GcdQuot(&gcd, &quot1, &quot2, r1, r2)` – procedure computes `gcd` and `quot1=r1/gcd` and `quot2=r2/gcd`, here `r1` and `r2` must be `RingElem`.

Queries

CoCoALib offers functions for querying various properties of `RingElem`s, and about relationships between `RingElem`s.

Let `r1` and `r2` be a (possibly const) `RingElem`s, and let `N` be a variable of type `BigInt` (Sec.8), and `q` a variable of type `BigRat` (Sec.9)

- `owner(r1)` – the ring to which `r1` is associated
- `IsZero(r1)` – true iff `r1` is zero
- `IsOne(r1)` – true iff `r1` is one
- `IsMinusOne(r1)` – true iff `-r1` is one
- `IsInvertible(r1)` – true iff `r1` has a multiplicative inverse
- `IsZeroDivisor(r1)` – true iff `r1` is zero-divisor
- `IsDivisible(r1, r2)` – true iff `r1` is divisible by `r2`
- `IsInteger(N, r1)` – true iff `r1` is the image of an integer (if true, a preimage is placed in `N`, otherwise it is left unchanged)
- `IsRational(q, r1)` – true iff `r1` is the image of a rational (if true, a preimage is placed in `q`, otherwise it is left unchanged)
- `IsDouble(d, r1)` – true iff `r1` is the image of a rational whose approx is put into d (false if overflow and `d` unchanged)

Note that `IsDivisible` tests divisibility in the ring containing the values: so 1 is not divisible by 2 in `RingZZ` (Sec.79), but their images in `RingQQ` (Sec.76) are divisible.

Ordering

If the ring is an ordered domain then these functions may also be used. You can discover whether CoCoALib thinks that the ring `R` is arithmetically ordered by calling `IsOrderedDomain(R)`: the value is `true` iff `R` is arithmetically ordered.

Note that comparison operations between `RingElem`s will fail if they do not belong to the same ring (the exception has code `ERR::MixedRings`). You may perform comparisons between a `RingElem` and an integer or a rational – the integer/rational is automatically mapped into the same ring as the `RingElem`.

Let `r1` and `r2` belong to an ordered ring. Trying to use any of these functions on elements belonging to a ring which is not ordered will produce an exception with code `ERR::NotOrdDomain`.

- `sign(r1)` – value is -1, 0 or +1 according as `r1` is negative, zero or positive
- `abs(r1)` – absolute value of `r1`
- `floor(r1)` – greatest integer `<= r1`
- `ceil(r1)` – least integer `>= r1`
- `NearestInteger(r1)` – returns nearest integer (`BigInt` (Sec.8)) to `r1`, halves round towards +infinity.
- `cmp(r1, r2)` – returns a value <0, $=0$, >0 according as `r1-r2` is <0, $=0$, >0
- `CmpDouble(r1, z)` – compare a ring elem with a `double`, result is <0, $=0$, >0 according as `r1-z` is <0, $=0$, >0
- `r1 < r2` – standard inequalities
- `r1 > r2` – ...
- `r1 <= r2` – ...
- `r1 >= r2` – ...

More operations on RingElems of a FractionField

If `owner(r)` is a fraction field then the following functions may be used. You can find out whether CoCoALib thinks that the ring `R` is a fraction field by calling `IsFractionField(R)`: the result is `true` iff `R` is a fraction field.

Let `K` denote a `FractionField` (Sec.32) Let `r` denote an element of `K`.

- `num(r)` – gives a copy of the numerator of `r` as an element of `BaseRing(K)`
- `den(r)` – gives a copy of the denominator of `r` as an element of `BaseRing(K)`

Note: the numerator and denominator are defined only upto multiples by a unit: so it is (theoretically) possible to have two elements of `FrF` which are equal but which have different numerators and denominators, for instance, (x-1)/(x-2) = (1-x)/(2-x)

More operations on RingElems of a QuotientRing

If `owner(r)` is a quotient ring then the following function may be called. You can find out whether CoCoALib thinks that the ring `R` is a quotient ring by calling `IsQuotientRing(R)`: the result is `true` iff `R` is a quotient ring.

In addition to the standard `RingElem` operations, elements of a `QuotientRing` (Sec.65) may be used in other functions.

Let `RmodI` denote a quotient ring. Let `r` denote a non-const element of `RmodI`.

- `CanonicalRepr(r)` – produces a `RingElem` (Sec.71) belonging to `BaseRing(RmodI)` whose image under `QuotientingHom(RmodI)` is `r`. For instance, if `r` = -3 in ZZ/(10) then this function could give 7 as an element of `RingZZ` (Sec.79).

More operations on RingElems of a RingTwinFloat

You can determine if an element belongs to a *twin-float* ring by calling `IsRingTwinFloat(owner(r))`: this yields `true` iff `r` belongs to a twin-float ring.

Let `x`, `y` be `RingElem` belonging to a `RingTwinFloat`

- `DebugPrint(out, x)` – print out both components of `x`
- `IsPracticallyEqual(x, y)` – returns true if `IsZero(x-y)` otherwise false.

In contrast the test `x==y` may throw a `RingTwinFloat::InsufficientPrecision` while `IsPracticallyEqual` will never throw this exception. `IsPracticallyEqual` is intended for use in a termination criterion for an iterative approximation algorithm (*e.g.* see `test-RingTwinFloat4.C`).

More operations on RingElems of a PolyRing

You can determine whether an element belongs to a *PolyRing* by calling `IsPolyRing(owner(r))`: the result is `true` iff `r` belongs to a poly ring.

Let `P` denote a polynomial ring. Let `f` denote a non-const element of `P`. Let `f1`, `f2` denote const elements of `P`. Let `v` denote a const vector of elements of `P`.

- `IsMonomial(f)`; – true iff `f` is non zero and of the form coeff*pp
- `AreMonomials(v)`; – true iff `v` is non zero and of the form coeff*pp (if `v` is empty it returns true)
- `IsConstant(f)`; – true iff `f` is "constant", i.e. the image of an element of the coeff ring.
- `IsIndet(f)`; – equivalent to `f` == x[i] for some index i
- `IsIndet(index, f)`; – equivalent to `f` == x[i]; and sets index = i
- `IsIrred(f)` – true iff `f` is irreducible in P
- `owner(f1)` – the owner of `f` as a `ring` (Sec.69); NB to get the owner as a `PolyRing` (Sec.63) use `AsPolyRing(owner(f1))`.
- `NumTerms(f1)` – the number of terms in `f1`.
- `StdDeg(f1)` – the standard degree of `f1` (deg(x[i])=1); **error** if `f1` is 0.
- `deg(f1)` – same as `StdDeg(f1)`.
- `MaxExponent(f1, var)` – maximum exponent of `var`-th indet in `f1` where `var` is the index of the indet in P (result is of type long).
- `LC(f1)` – the leading coeff of `f1`; it is an element of CoeffRing(P).
- `content(f1)` – gcd of the coeffs of `f1`; it is an element of CoeffRing(P). (content of zero poly is zero; if coeffs are in a field the content is 0 or 1)
- `CommonDenom(f1)` – the simplest common denominator for the coeffs of `f1`; it is an element of BaseRing(AsFractionField(CoeffRing(P))). (CoeffRing must be a FractionField of a GCD domain, otherwise **error**)
- `ClearDenom(f1)` – `f1*CommonDenom(f1)` (same restrictions as above)
- `deriv(f1, var)` – formal derivative of `f1` wrt. indet having index `var`.
- `deriv(f1, x)` – derivative of `f1` w.r.t. `x`, `x` must be an indeterminate (also works for f1 in `FractionField` of a `PolyRing`)

NOTE: to compute the *weighted degree* of a polynomial use the function `wdeg` defined for `RingElem` of a `SparsePolyRing` (Sec.87) (see below).

More operations on RingElems of a SparsePolyRing

You can determine whether an element belongs to a *sparse poly ring* by calling `IsSparsePolyRing(owner(r))`: the result is `true` iff `r` belongs to a poly ring.

Let `P` denote a `SparsePolyRing`. Let `f` denote a non-const element of `P`. Let `f1`, `f2` denote const elements of `P`. Let `expv` be a `vector<long>` of size equal to the number of indeterminates.

- `owner(f1)` – the owner of `f1` as a ring; NB to get the owner as a `SparsePolyRing` (Sec.87) use `AsSparsePolyRing(owner(f1))`
- `NumTerms(f1)` – the number of terms in `f1` with non-zero coefficient.
- `UnivariateIndetIndex(f)` – if `f` is univariate in j-th indet returns j, o/w returns -1
- `LPP(f1)` – the leading PP of `f1`; it is an element of PPM(P). Also known as *LT(f)* or *in(f)*
- `LF(f1)` – the leading form of `f1`; sum of all summands of highest weighted `degree` (Sec.17)

- `wdeg(f1)` – the weighted `degree` (Sec.17) of the leading PP of `f1` (see [KR] Sec.4.3); **error** if `f1` is 0. **NB** result is of type `CoCoA::degree` (see `degree` (Sec.17)). (contrast with `StdDeg(f1)` and `deg(f1)` defined for general `PolyRing` (Sec.63))
- `CmpWDeg(f1, f2)` – compare the weighted degrees of the LPPs of `f1` and `f2`; result is <0 $=0$ >0 according as deg(`f1`) $<$ $=$ $>$ deg(`f2`)
- `IsHomog(f)` – says whether `f` is homogeneous wrt weighted `degree` (Sec.17).
- `homog(f, h)` – returns `f` homogenized with indet `h` (requires `GrDim`=1 and `wdeg(h)`=1)
- `NR(f, v)` – returns the (normal) remainder of the Division Algorithm by `v`. If `v` is a GBasis this is the Normal Form
- `monomial(P,c,pp)` – returns `c*pp` as an element of `P` where `c` is in `CoeffRing(P)` and `pp` is in `PPM(P)`.
- `monomial(P,c,expv)` – returns `c*x[0]^exps[0]*x[1]^exps[1]*...` where `c` is in `CoeffRing(P)`, and x[i] are the indets of `P`.

Let `X` be an indet (*i.e.* a `RingElem` in `P`) or a vector of indices (`vector<long>`)

- `ContentWRT(f, X)` – the content of `f` wrt the indet(s) `X`; result is a `RingElem` in `P`
- `CoefficientsWRT(f, X)` – returns a `vector<CoeffPP>`: each `CoeffPP` has fields `myCoeff` and `myPP` where `myCoeff` is an element of `P` and `myPP` is in `PPM(P)` being a power product of the indets in `X`; the entries are in **decreasing order** of `myPP`.
- `CoeffVecWRT(f, x)` – `x` must be an indet; returns a `vector<RingElem>` whose k-th entry contains the coeff of `x^k` as an element of `P`; NB the coeff may be zero!

NB For running through the summands (or terms) of a polynomial use `SparsePolyIter`s (see `SparsePolyRing` (Sec.87)).

We have still doubts on the usefulness of these two functions:

- `CmpWDegPartial(f1, f2, i)` – compare the first `i` weighted degrees of the LPPs of `f1` and `f2`; result is <0 $=0$ >0 according as `deg(f1)` $<$ $=$ $>$ `deg(f2)`
- `IsHomogPartial(f,i)` – says whether `f` is homogeneous wrt the first `i` components of the weighted degree

Use the following two functions with great care: they throw an **error** if the `PPOrdering` (Sec.60) is not respected: (the coefficient `c` may be 0)

- `PushFront(f, c, expv)`– add to `f` the term `c`*t where t is the PP with exponent vector `expv`, and **assuming** that t $>$ `LPP(f)` or `f`==0
- `PushBack(f, c, expv)` – add to `f` the term `c`*t where t is the PP with exponent vector `expv`, and **assuming** that t $<$ t' for all t' appearing in `f`.

The corresponding member functions `myPushFront/myPushBack` will not check the validity of these assumpions: they should have a `CoCoA_ASSERT` to check in DEBUG mode.

More operations on RingElems of a DenseUPolyRing

You can determine whether an element belongs to a *DenseUPolyRing* by calling `IsDenseUPolyRing(owner(r))`: the result is `true` iff `r` belongs to a poly ring.

Let `P` denote a `DenseUPolyRing`. Let `f` denote an element of `P`.

- `monomial(P,c,exp)` – `c`*x^`exp` as an element of `P` with `c` an integer or in `CoeffRing(P)` `exp` a `MachineInt` (Sec.46)
- `coeff(f,d)` – the `d`-th coefficient of `f` (as a `ConstRingElem`, read-only)

WARNING Use this functions with great care: no checks on size and degree

Let `f` denote a *non-const* element of `P`.

- `myAssignCoeff(f,c,d)` – assigns the `d`-th coefficient in `f` to `c`
- `myAssignZeroCoeff(f,d)`
- `myAssignNonZeroCoeff(f,c,d)`

71.2.3 Notes on operations

Operations combining elements of different rings will cause a run-time **error**.

In all functions involving two `RingElem`s either `r1` or `r2` may be replaced by a machine integer, or by a big integer (an element of the class `BigInt` (Sec.8)). The integer value is automatically mapped into the ring owning the `RingElem` in the same expression.

The exponent `n` in the power function may be zero or negative, but a run-time error will be signalled if one attempts to compute a negative power of a non-invertible element or if one attempts to raise zero to the power zero. **NB** You cannot use `^` to compute powers – see *Bugs* section.

An attempt to perform an inexact division or to compute a GCD not in a GCD domain will produce a run-time error.

The printing of ring elements is always in decimal regardless of the `ostream` settings (this is supposed to be a feature rather than a bug).

At this point, if you are new to CoCoALib, you should probably look at some of the example programs in the `examples/` directory.

71.2.4 Writing functions with RingElems as arguments

One would normally expect to use the type `const RingElem&` for read-only arguments which are `RingElem`s, and `RingElem&` for read-write arguments. Unfortunately, doing so would lead to problems with the CoCoA library. **INSTEAD** you should use the types:

`ConstRefRingElem x`	for read-only arguments: morally `const RingElem& x`
`RingElem& x`	for read-write arguments
`RingElem x`	for read-only arguments which make a local copy

If you are curious to know why this non-standard quirk has to be used, read on.

When accessing matrix elements or coefficients in a polynomial CoCoALib uses *proxies*: these are objects which should behave much like `const RingElem` values. To allow easy use of such proxies in functions which want a read-only `RingElem` we use the type `ConstRefRingElem` (which is actually `const RingElemAlias&`) for the formal parameter.

Internally, ring element values are really smart pointers to the true value. Now the `const` keyword in C++ when applied to a pointer makes the pointer const while the pointed-to value remains alterable – this is not the behaviour we want for `const RingElem&`. To get the desired behaviour we have to use another type: the type we have called `ConstRefRingElem`.

71.2.5 ADVANCED USE OF RingElem

The rest of this section is for more advanced use of `ring` (Sec.69)s and `RingElem`s (*e.g.* by CoCoA library contributors). If you are new to CoCoA, you need not read beyond here.

Fast and Ugly Code

WE DO NOT RECOMMEND that you use what is described in this section. If you are curious to know a bit more how rings are implemented, you might find this section informative.

`RingElem`s are designed to be easy and pleasant to use, but this convenience has a price: a run-time performance penalty (and a memory space penalty too). Both penalities may be avoided by using *raw values* but at a considerable loss of programming convenience and safety. You should consider using raw values only if you are desperate for

speed; even so, performance gains may be only marginal except perhaps for operations on elements of a simple ring (e.g. a small finite field).

A `RingElem` object contains within itself an indication of the owning ring, and a *raw value* which is a pointer to where the real representation of the ring element value lies. These raw values may be accessed via the *raw* function. They may be combined arithmetically by calling member functions of the owning ring. For instance, if x,y,z are all RingElem objects all BELONGING TO EXACTLY THE SAME RING then we can achieve

```
x = y+z;
```

slightly faster by calling

```
owner(x)->my.Add(raw(x), raw(y), raw(z));
```

It should now be clear that the syntax involved is cumbersome and somewhat obscure. For the future maintainability of the code the simpler `x = y+z;` has many advantages. Furthermore, should x,y,z somehow happen not all to lie in the same ring then `x = y+z;` will act in a reasonable way, whereas the supposedly faster call will likely lead to many hours of debugging grief. The member functions for arithmetic (e.g. `myAdd`) **DO NOT PERFORM** sanity checks on their arguments: e.g. attempting to divide by zero could well crash the program.

If you use a *debugging version* of the CoCoA Library then some member functions do use assertions to check their arguments. This is useful during development, but must not be relied upon since the checks are absent from the *non-debugging version* of the CoCoA Library. See the file config.txt for more information.

This fast, ugly, unsafe way of programming is made available for those who desperately need the speed. If you're not desperate, don't use it!

Fast, Ugly and Unsafe operations on raw values

Read the section *Fast and Ugly Code* before using any of these!

Let `r` be a non-const raw value (*e.g.* `raw(x)`, with `x` a `RingElem`), and `r1`, `r2` potentially const raw values. Assume they are all owned by the ring `R`. Then the functions available are:

- `R->myNew()` – *construct* a new element of R, value=0
- `R->myNew(n)` – *construct* a new element of R, value=n
- `R->myNew(N)` – *construct* a new element of R, value=N
- `R->myNew(r1)` – *construct* a new element of R, value=r1
- `R->myDelete(r)` – *destroy* r, element of R (frees resources)
- `R->mySwap(r, s)` – swaps the two values (s is non-const raw value)
- `R->myAssignZero(r)` – r = 0
- `R->myAssign(r, r1)` – r = r1
- `R->myAssign(r, n)` – r = n (n is a long)
- `R->myAssign(r, N)` – r = n (N is a `BigInt` (Sec.8))
- `R->myNegate(r, r1)` – r = -r1
- `R->myAdd(r, r1, r2)` – r = r1+r2
- `R->mySub(r, r1, r2)` – r = r1-r2
- `R->myMul(r, r1, r2)` – r = r1*r2
- `R->myDiv(r, r1, r2)` – r = r1/r2 (division must be exact)
- `R->myIsDivisible(r, r1, r2)` – r = r1/r2, and returns true iff division was exact
- `R->myIsUnit(r1)` – IsUnit(r1)
- `R->myGcd(r, r1, r2)` – r = gcd(r1, r2)

- `R->myLcm(r, r1, r2)` – r = lcm(r1, r2)
- `R->myPower(r, r1, n)` – r = power(r1, n) BUT n MUST be non-negative!!
- `R->myIsZero(r1)` – r1 == 0
- `R->myIsZeroAddMul(r, r1, r2)` – ((r += r1*r2) == 0)
- `R->myIsEqual(r1, r2)` – r1 == r2
- `R->myIsPrintAtom(r1)` – true iff `r1` does not need brackets when a num or denom of a fraction
- `R->myIsPrintedWithMinus(r1)` – true iff the printed form of `r1` begins with a minus sign
- `R->myOutput(out, r1)` – out << r1
- `R->mySequentialPower(r, r1, n)` – normally it is better to use R->myPower(r, r1, n)
- `R->myBinaryPower(r, r1, n)` – normally it is better to use R->myPower(r, r1, n)

71.3 Maintainer documentation

(**NB** consider consulting also `QuotientRing` (Sec.65), `FractionField` (Sec.32) and `PolyRing` (Sec.63))

The design underlying rings and their elements is more complex than I would have liked, but it is not as complex as the source code may make it appear. The guiding principles are that the implementation should be flexible and easy/pleasant to use while offering a good degree of safety; extreme speed of execution was not a goal (as it is usually contrary to good flexibility) though an interface offering slightly better run-time efficiency remains.

Regarding flexibility: in CoCoALib we want to handle polynomials whose coefficients reside in (almost) any commutative ring. Furthermore, the actual rings to be used will be decided at run-time, and cannot restricted to a given finite set. We have chosen to use C++ inheritance to achieve the implementation: the abstract class `RingBase` defines the interface that every concrete ring class must offer.

Regarding ease of use: since C++ allows the common arithmetic operators to be overloaded, it is essential that these work as expected for elements of arbitrary rings – with the caveat that `/` means exact division, being the only reasonable interpretation. Due to problems of ambiguity arithmetic between elements of different rings is forbidden: e.g. let f in Q[x,y] and g in Z[y,x], where should f+g reside?

The classes in the file ring.H are closely interrelated, and there is no obvious starting point for describing them – you may find that you need to read the following more than once to comprehend it. Here is a list of the classes:

`ring`	value represents a ring; it is a smart pointer
`RingBase`	abstract class *defining what a ring is*
`RingElem`	value represents an element of a ring
`RingElemAlias`	reference to a `RingElem` belonging to someone else
`ConstRefRingElem`	C++ const-reference to a `RingElemAlias`
`RingElemConstRawPtr`	raw pointer to a *const* ring value
`RingElemRawPtr`	raw pointer to a ring value

For the first two see `ring` (Sec.69).

The classes `RingElem` and `RingElemAlias` are related by inheritance: they are very similar but differ in one important way. The base class `RingElemAlias` defines the data members which are inherited by `RingElem`. The essential difference is that a `RingElem` owns the value whereas a `RingElemAlias` does not. The two data members are `myR` and `myRawValue`: the first is the identity of ring to which the element belongs, and the second is the value in that ring (the value is stored in a format that only the owning ring can comprehend). All operations on ring elements are effected by member functions of the ring to which the value belongs.

The differing ownership inherent in `RingElemAlias` and `RingElem` lead to several consequences. The destructor of a `RingElem` will destroy in the internal representation of the value; in contrast, the destructor of a `RingElemAlias` does nothing. A `RingElemAlias` object becomes meaningless (& dangerous) if the owner of the value it aliases is destroyed.

Why did I create `RingElemAlias`? The main reason was to allow matrices and iterators of polynomials to be implemented cleanly and efficiently. Clearly a `matrix` (Sec.47) should be the owner of the values appearing as its entries, but we also want a way of reading the matrix entries without having to copy them. Furthermore, the matrix can use a compact representation: the ring to which its elements belong is stored just once, and not once for each element. Analogous comments apply to the coefficients of a polynomial.

As already stated above, the internal data layouts for objects of types `RingElem` and `RingElemAlias` are identical – this is guaranteed by the C++ inheritance mechanism. The subfield indicating the ring to which the value belongs is simply a `ring`, which is just a reference counting smart pointer. The subfield indicating the value is a raw pointer of type `void*`; however, when the raw pointer value is to be handled outside a ring element object then it is wrapped up as a `RingElemRawPtr` or `RingElemConstRawPtr` – these are simply wrapped copies of the `void*`.

The classes `RingElemRawPtr` and `RingElemConstRawPtr` are used for two reasons. One is that if a naked `void*` were used outside the ring element objects then C++ would find the call `RingElem(R,0)` ambiguous because the constant `0` can be interpreted either as an integer constant or as a null pointer: there are two constructors which match the call equally well. The other reason is that it discourages accidentally creating a ring element object from any old pointer; it makes the programmer think – plus I feel uneasy when there are naked `void*` pointers around. Note that the type of the data member `RingElemConstRawPtr::myPtr` is simply `void*` as opposed to `void const*` which one might reasonably expect. I implemented it this way as it is simpler to add in the missing constness in the member function `RingElemConstRawPtr::myRawPtr` than it would be to cast it away in the `myRawPtr` function of `RingElemRawPtr`.

Further comments about implementation aspects of the above classes.

The class `RingBase` declares a number of pure virtual functions for computing with ring elements. Since these functions are pure they must all be fully defined in any instantiable ring class (e.g. `RingZZImpl` or `RingFpImpl`). These member functions follow certain conventions:

RETURN VALUES: most arithmetic functions return no value, instead the result is placed in one of the arguments (normally the first argument is the one in which the result is placed), but functions which return particularly simple values (e.g. booleans or machine integers) do indeed return the values by the usual function return mechanism.

ARG TYPES: ring element values are passed as *raw pointers* (i.e. a wrapped `void*` pointing to the actual value). A read-only arg is of type `RingElemConstRawPtr`, while a writable arg is of type `RingElemRawPtr`. When there are writable args they normally appear first. For brevity there are typedefs `ConstRawPtr` and `RawPtr` in the scope of `RingBase` or any derived class.

ARG CHECKS: sanity checks on the arguments are NOT CONDUCTED (e.g. the division function assumes the divisor is non-zero). These member functions are supposed to be fast rather than safe.

In a few cases there are non-pure virtual member functions in `RingBase`. They exist either because there is a simple universal definition or merely to avoid having to define inappropriate member functions (*e.g.* gcd functions when the ring cannot be a gcd domain). Here is a list of them:

- `myIsUnit(x)` – default checks that 1 is divisible by `x`
- `myGcd(lhs, x, y)` – gives an error: either `NotGcdDom` or `NYI`
- `myLcm(lhs, x, y)` – gives an error: either `NotGcdDom` or `NYI`
- `myGcdQuot(lhs, xquot, yquot, x, y)` – gives an error: either `NotGcdDom` or `NYI`
- `myExgcd(lhs, xcofac, ycofac, x, y)` – gives an error: either `NotGcdDom` or `NYI`
- `myIsPrintAtom(x)` – defaults to false
- `myIsPrintedWithMinus(x)` – gives `SERIOUS` error
- `myIsMinusOne(x)` – defaults to myIsOne(-x); calculates -x
- `myIsZeroAddMul(lhs, y, z)` – computes lhs += y*z in the obvious way, and calls `myIsZero`
- `myCmp(x, y)` – gives `NotOrdDom` error
- `mySign(x)` – simply calls `myCmp(x, 0)`, then returns -1,0,1 accordingly

  There are three non-virtual member functions for calculating powers: one uses the sequential method, the other two implement the repeated squaring method (one is an entry point, the other an implementation detail). These are non-virtual since they do not need to be redefined; they are universal for all rings.

  For the moment I shall assume that the intended meaning of the pure virtual functions is obvious (given the comments in the source code).

Recall that arithmetic operations on objects of type `ConstRefRingElem` (which matches `RingElem` too) are converted into member function calls of the corresponding owning ring. Here is the source code for addition of ring elements – it typifies the implementation of operations on ring elements.

```
RingElem operator+(ConstRefRingElem x, ConstRefRingElem y)
{
  const ring& Rx = owner(x);
  const ring& Ry = owner(y);
  if (Rx != Ry)
    error(CoCoAError(ERR::MixedRings, "RingElem + RingElem"));

  RingElem ans(Rx);
  Rx->myAdd(raw(ans), raw(x), raw(y));
  return ans;
}
```

The arguments are of type `ConstRefRingElem` since they are read-only, and the return type is `RingElem` since it is new self-owning value (it does not refer to a value belonging to some other structure). Inside the function we check that the rings of the arguments are compatible, and report an error if they are not. Otherwise a temporary local variable is created for the answer, and the actual computation is effected via a member function call to the ring in which the values lie. Note the use of the `raw` function for accessing the raw pointer of a ring element. In summary, an operation on ring elements intended for public use should fully check its arguments for compatibility and correctness (*e.g.* to avoid division by zero); if all checks pass, the result is computed by passing raw pointers to the appropriate member functions of the ring involved – this member function assumes that the values handed to it are compatible and valid; if not, *undefined behaviour* will result (*i.e.* a crash if you are lucky).

Most of the member functions of a ring are for manipulating raw values from that same ring, a few permit one to query properties of the ring. The type of a raw value is `RingBase::RawValue`, which helpfully abbreviates to RawValue inside the namespace of `RingBase`. Wherever possible the concrete implementations should be **exception safe**, *i.e.* they should offer either the strong exception guarantee or the no-throw guarantee (according to the definitions in *Exceptional C++* by Sutter).

71.4 Bugs, Shortcomings and other ideas

I have chosen not to use `operator^` for computing powers because of a significant risk of misunderstanding between programmer and compiler. The syntax/grammar of C++ cannot be changed, and `operator^` binds less tightly than (binary) `operator*`, so any expression of the form `a*b^c` will be parsed as `(a*b)^c`; this is almost certainly not what the programmer intended. To avoid such problems of misunderstanding I have preferred not to define `operator^`; it seems too dangerous.

Note about comparison operators ($<, <=, >, >=$, and !=). The C++ STL does have templates which will define all the relational operators efficiently assuming the existence of `operator<` and `operator==`. These are defined in the namespace `std::rel_ops` in the standard header file `<utility>`. I have chosen NOT to use these because they can define only *homogeneous* comparisons; so the comparisons between `ConstRefRingElem` and `int` or `BigInt` (Sec.8) would still have to be written out manually, and I prefer the symmetry of writing them all out. See p.69ff of Josuttis for details.

The function `myAssignZero` was NECESSARY because `myAssign(x, 0)` was ambiguous (ambiguated by the assignment from an `mpz_t`). It is no longer necessary, but I prefer to keep it (for the time being).

The requirement to use the type `ConstRefRingElem` for function arguments (which should normally be `const RingElem&` is not ideal, but it seems hard to find a better way. It is not nice to expect users to use a funny type for their function arguments. How else could I implement (noncopying) access to coefficients in a polynomial via an iterator, or access to matrix elements?

Would we want ++ and – operators for `RingElem`s???

Should (some of) the query functions return `bool3` values? What about properties which are hard to determine?

How to generate random elements from a ring?

Anna thinks that `NearestInteger` could handle specially elements of `RingZZ` (Sec.79) rather than doing the full wasteful computation. Not sure if the extra code and complication would really make a difference in practice.

`gcd` and `lcm`: there is no guarantee on sign/monic because it may be costly to compute and generally useless.

71.5 Main changes

2013

- May (v0.9953):
  - added IsZeroDivisor -

72 RingFp (John Abbott)

72.1 User documentation for the class RingFpImpl

The usual way to perform arithmetic in a (small, prime) finite field is to create the appropriate ring via the pseudo-constructors `NewZZmod` (or `NewQuotientRing` if you prefer) which are documented in `QuotientRing` (Sec.65). These functions will automatically choose a suitable underlying implementation, and you should normally use them.

In some special circumstances, you may wish to choose explicitly the underlying implementation. CoCoALib offers three distinct implementations of small prime finite fields: `RingFp` (described here), and `RingFpLog` (Sec.74) and `RingFpDouble` (Sec.73). Of these `RingFp` is probably simplest and fastest implementation – this file describes how to create a `RingFp` implementation.

To create a `ring` (Sec.69) of this specific type use one of the pseudo-constructors:

```
NewRingFp(p) -- p a machine integer or BigInt
NewRingFp(I) -- I an ideal of RingZZ
NewRingFp(p, res) -- p a machine integer, res is either ``GlobalSettings::SymmResidues`` or ``GlobalSettin
```

These pseudo-constructors are for creating small prime finite fields; they will fail if the characteristic is not prime or is too large: the error signalled by throwing a `CoCoA::ErrorInfo` whose *code* is `CoCoA::ERR::BadSmallFpChar`. You can test whether an argument is suitable by calling `IsGoodForRingFp`.

The default convention for printing residues is specified when you create the `GlobalManager` (Sec.37); you can also specify explicitly which convention to use by giving a second argument to the pseudo-ctor `NewRingFp`. Note that the **internal representation** is always least non-negative regardless of the output convention chosen.

If you seek a means for fast arithmetic in small finite fields consult the documentation about `SmallFpImpl` (Sec.82), `SmallFpLogImpl` (Sec.83), and `SmallFpDoubleImpl` (Sec.81). All arithmetic on elements of a `RingFp` is actually carried out by a `SmallFpImpl` object.

NewZZmod or NewRingFp?

If `n` is a small prime then `NewZZmod(n)` produces the same result as `NewRingFp(n)` (or perhaps `NewRingFpDouble(n)`). If `n` is not a small prime then `NewRingFp(n)` throws an exception whereas `NewZZmod(n)` will produce a working quotient ring.

72.1.1 Examples

- ex-RingFp1.C
- ex-RingFp2.C

72.2 Maintainer documentation for the class RingFpImpl

The class `RingFpImpl` is a low-level implementation of (small prime) finite fields; it is not intended for direct use by casual CoCoA library users.

The class `RingFpImpl` is intended to implement small, prime finite fields. The constructor is more complicated than one might expect, this is because the `RingFpImpl` object must store a little extra information to fulfil its role as a `QuotientRingBase`. Currently, the characteristic must be prime (otherwise it wouldn't be a field) and must also be small enough that its square fits into a `SmallFpElem_t` (probably `unsigned long`, see the file config.H); if not, an error is signalled.

Extreme efficiency is NOT one of the main features of this version; contrast this with `SmallFpImpl`.

The class RingFpImpl derives from QuotientRingBase, which in turn is derived from RingBase: see QuotientRing (Sec.65) and ring (Sec.69) for more details. Note that there is no RingFp class; a RingFpImpl object can only be accessed as a QuotientRing (Sec.65).

Note the use of "argument checking" static member functions in the ctor: this is because const data members must be initialized before the main body of the ctor is entered.

A member typedef RingFpImpl::value_t specifies the type used for representing the value of an element of a RingFpImpl: this is a typedef for SmallFpElem_t which is defined in config.H (to facilitate tuning for different platforms).

The data members are those of a QuotientRingBase (which are used only for answering queries about a QuotientRing (Sec.65)), plus the characteristic of the field (held as an value_t in myModulus), and an auto-pointer to a copy of the zero and one elements of the ring.

The zero and one elements of the ring is held in an auto_ptr<> for consistency with the implementation of other rings – in this simple class it is not really necessary for exception safety.

This implementation is very simplistic: almost every operation is delegated to the class SmallFpImpl. The implementation class has been separated so that its inline member functions can be used directly by some other special case code (*e.g.* polynomials with SmallFp coeffs). See SmallFpImpl (Sec.82) for details. I note that the residues are represented internally as the least non-negative value in the residue class regardless of the user's choice of type of residue.

The largest permitted modulus for a RingFpImpl may depend on the platform. On a 32-bit machine the modulus must surely be less than 65536 – refer to SmallFpImpl (Sec.82) for details. A 64-bit machine may allow larger characteristics.

Although it may seem wasteful to use heap memory for the values of elements in a RingFpImpl, trying to make them "inline" leads to lots of problems. Originally we had implemented the values as "inline", and the resulting problems delayed CoCoALib by almost a year.

72.3 Bugs, shortcomings and other ideas

Why does the class RingFp not exist? Well, my current thoughts are that since a RingFp would not do anything special which a QuotientRing (Sec.65) cannot do, it seems needless extra complication to create a "useless" class. In particular, it cannot offer better run-time performance. If you want to compute quickly modulo a small prime you must use SmallFpImpl (Sec.82) directly.

Probably RingFp, RingFpLog (Sec.74) and RingFpDouble (Sec.73) could be replaced by instances of a template class – the template parameter would be SmallFpImpl, SmallFpLogImpl or SmallFpDoubleImpl accordingly.

Why do all the member functions blindly forward their calls to the SmallFpImpl member functions? This means that the error message for division by zero (say) will refer to SmallFpImpl rather than RingFpImpl. Does this really matter that much? Obviously the much same applies to RingFpLogImpl and RingFpDoubleImpl.

73 RingFpDouble (John Abbott)

73.1 User documentation for the class RingFpDoubleImpl

The usual way to perform arithmetic in a (small, prime) finite field is to create the appropriate ring via the pseudo-constructors NewZZmod (or NewQuotientRing if you prefer) which are documented in QuotientRing (Sec.65). These functions will automatically choose a suitable underlying implementation, and you should normally use them.

In some special circumstances, you may wish to choose explicitly the underlying implementation. CoCoALib offers three distinct implementations of small prime finite fields: RingFp (Sec.72), and RingFpLog (Sec.74) and RingFpDouble (Sec.73) (described here). Of these RingFpDouble may offer the highest limit on the characteristic (*e.g.* on 32-bit machines) – this file describes how to create a RingFpDouble implementation.

To create a ring (Sec.69) of this specific type use one of the pseudo-constructors:

```
NewRingFpDouble(p) -- p a machine integer or BigInt
NewRingFpDouble(I) -- I an ideal of Z
NewRingFpDouble(p, res) -- p a machine integer, res is either ``GlobalSettings::SymmResidues`` or ``Global
```

These pseudo-constructors are for creating small prime finite fields; they will fail if the characteristic is not prime or is too large: the error signalled by throwing a CoCoA::ErrorInfo whose *code* is CoCoA::ERR::BadSmallFpChar.

You can test whether an argument is suitable by calling `IsGoodFoRingFpDouble`.

In the directory `examples/` there is a small example program showing how small finite fields (with known implementation) can be created and used: `ex-RingFp2.C`.

The default convention for printing residues is specified when you create the `GlobalManager` (Sec.37); you can also specify explicitly which convention to use by giving a second argument to the pseudo-ctor `NewRingFp`. Note that the **internal representation** is always least non-negative regardless of the output convention chosen.

If you seek a means for fast arithmetic in small finite fields consult the documentation about `SmallFpImpl` (Sec.82), `SmallFpLogImpl` (Sec.83), and `SmallFpDoubleImpl` (Sec.81). All arithmetic on elements of a `RingFp` is actually carried out by a `SmallFpImpl` object.

73.2 Maintainer documentation for the class RingFpDoubleImpl

The class `RingFpDoubleImpl` is a low-level implementation of (small prime) finite fields; it is not intended for direct use by casual CoCoA library users. Internally values are represented using `double`s: this may permit a higher maximum characteristic on some computers (*e.g.* 32-bitters).

The class `RingFpDoubleImpl` is intended to represent small, prime finite fields. The constructor is more complicated than one might expect; this is because the `RingFpDoubleImpl` object must store a little extra information to fulfil its role as a `QuotientRing` (Sec.65). Currently, the characteristic must be prime (otherwise it wouldn't be a field). Furthermore, the characteristic p must also be small enough that all integers up to p*(p-1) can be represented exactly as `double`s. Creating a `RingFpDoubleImpl` takes almost constant time (except for the primality check). An error is signalled (*i.e.* a `CoCoA::ErrorInfo` is thrown) if the characteristic is too large or not prime.

Extreme efficiency is NOT one of the main features of this version: contrast with `SmallFpDoubleImpl` (Sec.81).

The class `RingFpDoubleImpl` derives from `QuotientRingBase`, which in turn is derived from `RingBase`: see `QuotientRing` (Sec.65) and `ring` (Sec.69) for more details. Note that there is no `RingFpDouble` class; a `RingFpDoubleImpl` object can only be accessed as a `QuotientRing`.

Note the use of "argument checking" static member functions in the ctor: this is because `const` data members must be initialized before the main body of the ctor is entered.

A member typedef specifies the type used internally for representing the value of an element of a `RingFpDoubleImpl`: currently this is just `SmallFpDoubleImpl::value_t` which is `double`.

Essentially all operations are delegated to the class `SmallFpDoubleImpl`. The two classes are separate so that the inline operations of `SmallFpDoubleImpl` can be accessed directly in certain other special case implementations (*e.g.* polynomials with coeffs in a small finite field). See the documentation on `SmallFpDoubleImpl` (Sec.81) for details.

The data members are those of a `QuotientRingBase` (which are used only for answering queries about a `QuotientRing`), plus the characteristic of the field (held as an `value_t` in `myModulusValue`), and an auto-pointer to a copy of the zero and one elements of the ring.

The zero and one elements of the ring is held in an auto_ptr<> for consistency with the implementation of other rings – in this simple class it is not really necessary for exception safety.

The largest permitted modulus for a `RingFpImpl` may depend on the platform. If IEEE doubles are used then moduli up to 67108859 are permitted – refer to `SmallFpDoubleImpl` (Sec.81) for details.

Although it may seem wasteful to use heap memory for the values of elements in a `RingFpDoubleImpl`, trying to make them "inline" leads to lots of problems – see `RingFp` (Sec.72) for more details.

73.3 Bugs, shortcomings and other ideas

Can reduction modulo p be made faster?

Run-time performance is disappointing.

I wonder if this code will ever prove useful to anyone.

74 RingFpLog (John Abbott)

74.1 User documentation for the class RingFpLogImpl

The usual way to perform arithmetic in a (small, prime) finite field is to create the appropriate ring via the pseudo-constructors `NewZZmod` (or `NewQuotientRing` if you prefer) which are documented in `QuotientRing` (Sec.65). These functions will automatically choose a suitable underlying implementation, and you should normally use them.

In some special circumstances, you may wish to choose explicitly the underlying implementation. CoCoALib offers three distinct implementations of small prime finite fields: `RingFp` (Sec.72), and `RingFpLog` (described here) and `RingFpDouble` (Sec.73). Of these `RingFpLog` may be the fastest if your processor has slow division instructions – this file describes how to create a `RingFpLog` implementation.

To create a `ring` (Sec.69) of this specific type use one of the pseudo-constructors:

```
NewRingFpLog(p) -- Z ring of integers, p a machine integer or BigInt
NewRingFpLog(I) -- Z ring of integers, I an ideal of Z
NewRingFpLog(p, res) -- p a machine integer, res is either ``GlobalSettings::SymmResidues`` or ``GlobalSet
```

These pseudo-constructors are for creating small prime finite fields; they will fail if the characteristic is not prime or is too large: the error signalled by throwing a `CoCoA::ErrorInfo` whose *code* is `CoCoA::ERR::BadSmallFpChar`. You can test whether an argument is suitable by calling `IsGoodFoRingFpLog`.

In the directory `examples/` there is a small example program showing how small finite fields (with known implementation) can be created and used: `ex-RingFp2.C`.

The default convention for printing residues is specified when you create the `GlobalManager` (Sec.37); you can also specify explicitly which convention to use by giving a second argument to the pseudo-ctor `NewRingFpLog`. Note that the **internal representation** is always least non-negative regardless of the output convention chosen.

If you seek a means for fast arithmetic in small finite fields consult the documentation about `SmallFpImpl` (Sec.82), `SmallFpLogImpl` (Sec.83), and `SmallFpDoubleImpl` (Sec.81). All arithmetic on elements of a `RingFp` is actually carried out by a `SmallFpImpl` object.

74.2 Maintainer documentation for the class RingFpLogImpl

The class `RingFpLogImpl` is a low-level implementation of (small prime) finite fields; it is not intended for direct use by casual CoCoA library users. Multiplication and division are effected using discrete log/exp tables.

The class `RingFpLogImpl` is intended to represent small, prime finite fields. The constructor is more complicated than one might expect, this is because the `RingFpLogImpl` object must store a little extra information to fulfil its role as a `QuotientRingBase`. Currently, the characteristic must be prime (otherwise it wouldn't be a field). Furthermore, the characteristic must also be less than 65536 even on machines with 64-bit arithmetic – larger values are prohibited as the internal tables would become excessively large. Creating a `RingFpLogImpl` of characteristic p takes time roughly linear in p; space consumption is linear in p. An error is signalled if the characteristic is too large or not prime.

Extreme efficiency is NOT one of the main features of this version.

The class `RingFpLogImpl` derives from `QuotientRingBase`, which in turn is derived from `RingBase`: see `QuotientRing` (Sec.65) and `ring` (Sec.69) for more details. Note that there is no `RingFpLog` class; a `RingFpLogImpl` object can only be accessed via a `QuotientRing`.

Note the use of "argument checking" static member functions in the ctor: this is because `const` data members must be initialized before the main body of the ctor is entered.

A member typedef `RingFpLogImpl::value_t` specifies the type used for representing the value of an element of a `RingFpLogImpl`: currently this is a typedef for `SmallFpLogElem_t` which is defined in config.H.

Essentially all operations are delegated to the class `SmallFpLogImpl`. The two classes are separate so that the inline operations of `SmallFpLogImpl` can be accessed directly in certain other special case implementations (*e.g.* polynomials with coeffs in a SmallFp). See the documentation on `SmallFpLogImpl` (Sec.83) for details. I note that the residues are represented as the least non-negative value in the residue class.

The data members are those of a `QuotientRingBase` (which are used only for answering queries about a `QuotientRing` (Sec.65)), plus the characteristic of the field (held as a `value_t` in `myModulusValue`), and an auto-pointer to a copy of the zero and one elements of the ring.

The zero and one elements of the ring is held in an auto_ptr< > for consistency with the implementation of other rings – in this simple class it is not really necessary for exception safety.

The largest permitted modulus for a `RingFpLogImpl` may depend on the platform. On a 32-bit machine the modulus must surely be less than 65536 – refer to `SmallFpLogImpl` (Sec.83) for details. A 64-bit machine may allow larger characteristics.

Although it may seem wasteful to use heap memory for the values of elements in a RingFpLogImpl, trying to make them "inline" leads to lots of problems – see `RingFp` (Sec.72) for more details

74.3 Bugs, shortcomings and other ideas

See also some comments in the "bugs" section of RingFp.txt.

The code is not very smart in the case of characteristic 2.

Run-time performance is disappointing.

I wonder if this code will ever prove useful to anyone.

75 RingHom (John Abbott)

75.1 User documentation for the files RingHom.H and RingHom.C

The class `RingHom` is intended to represent homomorphisms between rings. Currently there is no way to represent more general maps between rings. It is possible to create a *partial* homomorphism which can generate run-time errors when applied to certain values.

The main operations available for `RingHom`s are application to a `RingElem` (Sec.71), and composition: both operations use function application syntax (see below for details).

75.1.1 Examples

- ex-RingHom1.C
- ex-RingHom2.C
- ex-RingHom3.C
- ex-RingHom4.C
- ex-RingHom5.C

75.1.2 Constructors

Here is a complete list of pseudo-constructors for ring homomorphisms (some are defined in other files, *e.g.* `QuotientRing.H` or `FractionField.H`). You should check also the documentation for `CanonicalHom` (Sec.12) which tries to offer an easy way of building certain simple canonical homomorphisms.

- `IdentityHom(R)` – where `R` may be any `ring` (Sec.69), gives the identity homomorphism on `R`

Embeddings

- `ZZEmbeddingHom(R)` – gives the unique homomorphism from `RingZZ` (Sec.79) to the `ring` (Sec.69) `R`
- `QQEmbeddingHom(R)` – **may be partial** gives the unique homomorphism from `RingQQ` (Sec.76) to the `ring` (Sec.69) `R`. Note that the resulting homomorphism may be only partial (e.g. if Characteristic(R) is not 0, or if the codomain is not a field).
- `CoeffEmbeddingHom(P)` – where `P` is a `PolyRing` (Sec.63), gives the embedding homomorphism from the coefficient ring into the polynomial ring.
- `EmbeddingHom(FrF)` – where `FrF` is a `FractionField` (Sec.32), gives the embedding homomorphism from the base ring into the fracion field (i.e. x |-> x/1)

Induced homomorphisms

- `InducedHom(RmodI, phi)` – where `RmodI` is a `QuotientRing` (Sec.65), gives the homomorphism R/I –> S induced by `phi`: R –> S (which must have the base ring of `RmodI` as its domain, and whose kernel must contain the defining ideal of `RmodI`)
- `InducedHom(FrF, phi)` – **may be partial** where `FrF` is a `FractionField` (Sec.32), gives the homomorphism induced by `phi` (which must have the base ring of `FrF` as its domain). Note that the resulting homomorphism may be only partial (e.g. if ker(`phi`) is non-trivial, or if the codomain is not a field).

Homomorphisms on polynomial rings

- `PolyAlgebraHom(Rx, Ry, xImages)` – where
  - `Rx` and `Ry` are `PolyRing` (Sec.63) with the same CoeffRing
  - `xImages` is a `vector` of `RingElem` (Sec.71)

  gives the homomorphism from `Rx` to `Ry` mapping coefficients into themselves and mapping the k-th indeterminate of `Rx` to the k-th value in `xImages` (i.e. having index k-1)
- `PolyRingHom(Rx, S, CoeffHom, xImages)` – where
  - `Rx` is a `PolyRing` (Sec.63)
  - `CoeffHom` is a homomorphism whose domain is `CoeffRing(Rx)` and codomain is `S` or CoeffRing(S) (if `S` is a `PolyRing` (Sec.63))
  - `xImages` is a `vector` of `RingElem` (Sec.71)

  gives the homomorphism from `Rx` to `S` mapping coefficients according to `CoeffHom` and mapping the k-th indeterminate of `Rx` to the k-th value in `xImages` (i.e. having index k-1)

75.1.3 Applying a RingHom

A `RingHom` may applied using natural syntax: - let phi be an object of type `RingHom` - let x be an object of type `RingElem` (Sec.71) - let n be of type `long` or `int` - let N be an object of type `BigInt` (Sec.8) - let q be an object of type `BigRat` (Sec.9)

```
phi(x)  applies phi to x; error if owner(x) != domain(phi)
phi(n)  applies phi to the image of n in domain(phi)
phi(N)  applies phi to the image of N in domain(phi)
phi(q)  applies phi to the image of q in domain(phi)
```

In all cases the result is a `RingElem` (Sec.71) belonging to the codomain of phi. Currently *partial* homomorphisms are allowed, so applying a RingHom could trigger an error (*e.g.* an induced hom from Q to Z/(3) applied to 1/3).

If the `RingElem` (Sec.71) supplied belongs to the wrong `ring` (Sec.69) (*i.e.* not the domain of the `RingHom`) then an exception is thrown with code `ERR::BadRingHomArg1`. If the argument belongs to the right `ring` (Sec.69) but lies outside the domain then an exception with code `ERR::BadRingHomArg2` is thrown.

75.1.4 Composition

Two `RingHom`s may be composed using a fairly natural syntax: if we have two `RingHom`s phi:R -> S and theta:S -> T then their composition may be computed using the syntax

```
theta(phi)   the composite homomorphism "apply phi first then theta"
```

If the domain of the outer homomorphism is different from the codomain of the inner homomorphism then an exception is thrown with code `ERR::BadCompose`.

75.1.5 Domain and Codomain

We may ask for the domain and codomain of a `RingHom` phi:

```
domain(phi)       gives a const ref to the domain
codomain(phi)     gives a const ref to the codomain
```

Note that the domain and codomain are merely `ring` (Sec.69)s, they "forget" any special ring type (such as `SparsePolyRing` (Sec.87)).

75.1.6 Kernel

Currently it is not possible to ask for the kernel of a `RingHom`.

75.1.7 Member Functions for Operations on Raw Values

All operations on a `RingHom` are invisibly converted into member function calls on a `RingHomBase`. It is possible to call these member functions directly: the main difference is that the member functions do not perform any sanity checking on their arguments (so they should be slightly faster but if you hand in incompatible arguments, you'll probably get an ugly crash).

75.2 Maintainer documentation for the files RingHom.H and RingHom.C

These files contain two "generic" classes (`RingHom` and `RingHomBase`), and a trivial concrete class representing the identity ring homomorphism, `IdentityRingHom`. Most of this section is dedicated to the two generic classes since they represent the primary contribution to the CoCoA library.

The class `RingHom` is little more than a "reference counting smart pointer" class to objects of type `RingHomBase`; this latter type is designed to support intrusive reference counting. Beyond its role as a smart pointer `RingHom` offers four "function application" syntaxes:

```
RingElem RingHom::operator()(ConstRefRingElem x) const;
RingElem RingHom::operator()(long n) const;
RingElem RingHom::operator()(const BigInt& N) const;
RingHom RingHom::operator()(const RingHom&) const;
```

The first three support a natural syntax for applying the homomorphism to a ring element, a small integer, or a large integer. The last offers a fairly natural syntax for creating the composition of two homomorphisms.

The class `RingHomBase` is a purely abstract class which is used to specify the interface which any concrete ring homomorphism class must offer. In particular this base class already includes an intrusive reference counter, as required by `RingHom`. It also includes two private data members `myDomainValue` and `myCodomainValue` which store the domain and codomain rings. Note that these data fields are plain `ring` (Sec.69)s and so "forget" any special ring type which the domain or codomain may have had. Originally I had hoped to preserve any special ring type information, but this seemed to lead to a confusing and complex implementation (which probably would never have worked as I hoped). The two `ring` (Sec.69) fields may be read using the accessor functions:

```
const ring& myDomain() const;
const ring& myCodomain() const;
```

A concrete class implementing a ring homomorphism must supply definition for the following (pure virtual) functions:

```
virtual void myApply(RingBase::RawValue& image, RingBase::ConstRawValue arg) const;
virtual void myOutputSelf(std::ostream& out) const;
```

DO NOTE THAT the two arguments to `myApply` normally belong to DIFFERENT rings. `arg` belongs to `myDomain()` whereas `image` belongs to `myCodomain()`. The function `myOutputSelf` should print out a useful description of the homomorphism.

75.3 Bugs, Shortcomings and other ideas

Cannot compute a kernel of a RingHom.

Arranging for `domain(phi)` and `codomain(phi)` to preserve C++ type information about the respective rings (*e.g.* `PolyRing` (Sec.63) or `FractionField` (Sec.32) rather than simply `ring` (Sec.69)), appears to be difficult to achieve in any reasonable manner. I've decided that it is much simpler just to discard all special type information, and return simply `ring` (Sec.69)s. If the user knows something more, he can use a "cast" function like `AsFractionField`. Even if it were feasible to maintain such C++ type info, there would have to n-squared cases to cover all possible combinations of domain and codomain.

We should implement more special cases: *e.g.* same vars different coeff ring, PP –> PP, other... Also need some way of handling canonical homomorphisms.

Some special cases of homomorphic embeddings R –> S: (may belong with the special types of ring to which they are associated)

- (a) S is an identical copy of R
- (b) S is the same (poly)ring as R but with a different term ordering
- (c) R, S are the same polynomial ring (same vars and ordering) but with different coefficients
- (d) each generator of R maps to a power product (or 0) in S
- (e) S is the same as R but with more variables (is also of type (d))
- (f) permutation of the variables (is also of type (d))
- (g) general homomorphism mapping
- (h) S is the fraction field of R

75.4 Some very old notes about implementing rings

This all needs to be sorted out!

75.4.1 Mapping elements between rings automatically

How to decide whether a value can be mapped into the current_ring?

If the rings are marked as being equivalent isomorphically then we can just use the obvious isomorphism. A more interesting case is when a value resides in a ring which is a natural subring of the current_ring e.g. Z inside Q(sqrt(2))[x,y,z].

One could argue that to create `Q(sqrt(2))[x,y,z]` we had to follow this path

- `Z` –> fraction field `Q`
- `Q` –> polynomial ring (1 indet) or DUP extension `Q[gensym]`
- `Q[gensym]` –> quotient by gensym^2-2 to get `Q(sqrt(2))`
- `Q(sqrt(2))` –> polynomial ring (3 indets) `Q(sqrt(2))[x,y,z]`

From this it ought to be easy to identify natural embeddings of `Z`, `Q`, and (possibly) `Q(sqrt(2))` in `Q(sqrt(2))[x,y,z]`. We do not get an embedding for Q[gensym] since we had to generate the symbol *gensym* and no one else can create the same gensym. Because of this it is not altogether clear that an independently created copy of `Q(sqrt(2))` can be embedded automatically, since that copy would have a different symbol/gensym. Now if the algebraic extension were achieved directly...

Would we want `Q[x]/(x^2-2)` to be regarded as isomorphically equivalent to `Q[y]/(y^2-2)`? In fact there are two possible isoms: `x <---> y` and `x <---> -y`. I think that these should not be viewed as isom automatically, especially as there is more than one possible choice.

In contrast, if `R = Q[x]/(x^2-2)`, and `S = Q[x]/(36-18x^2)`, and `T = Q[x]/(x^2-2)`. It is clear that `Q[x]` can be mapped into each of `R`, `S` and `T` in a natural way. Of course, in each case `x` stands for sqrt(2), and it wouldn't be too hard to spot that `R` and `T` are *identical*; it is not quite as simple to see that `R` and `S` are isom. Presumably with a little more effort one could create examples where it could be jolly hard to spot that two such rings are just

the same ring. For this reason, I think no attempt should be made to spot such *natural isoms* between *independent* rings. Had `T` been created from `R` (e.g. by making copy via assignment) then they would no longer be independent, and a natural isom could be deduced automatically. Now I think about it, a facility to make a copy of a ring WITHOUT the natural isom should be made available.

There is also a need for a way to specify that one ring embeds naturally into another (and via which homomorphism), or indeed that they are isomorphic. Isomorphism could be expressed by giving two inverse homs – the system could then check that the homs are inverse on the generators, how it would check that the maps are homs is not so clear (perhaps the only maps which can be created are homs). Oooops, this would allow one to declare that `Z` and `Q` (or `Z[x]` and `Q[x]`) are isom..... need to think more about this!

A similar mechanism will be needed for modules (and vector spaces). A module should naturally embed into a vector space over the fraction field of the base ring....

Conceivably someone might want to change the natural embedding between two rings. So a means of finding out what the natural embedding is will be necessary, and also a way replacing it.

There is also a general question of retracting values into *subrings*. Suppose I have computed 2 in `Q(x)`, can I get the integer 2 from this? In this case I think the user must indicate explicitly that a retraction is to occur. Obviously retraction can only be into rings *on the way* to where the value currently resides.

Other points to note:

```
Q(x) = Z(x) = FrF(Z[x]) == FrF(FrF(Z)[x])
```

Q(alpha) = FrF(Z[alpha]) though denoms in Q(alpha) can be taken in Z

Q[alpha]/I_alpha = FrF(Z[alpha]/I_alpha) **BUT** the ideal on LHS is an ideal inside Q[alpha] whereas that on RHS is in Z[alpha]. Furthermore Z[alpha]/I_alpha is *hairy* if the min poly of alpha is not monic!

76 RingQQ (John Abbott, Anna M. Bigatti)

76.1 User documentation for RingQQ

The call `RingQQ()` produces the CoCoA `ring` (Sec.69) which represents QQ, the field of rational numbers. Calling `RingQQ()` several times will always produce the same unique CoCoA `ring` (Sec.69) representing QQ.

Strictly, there is a limit on the size of elements you can create, but the limit is typically high enough not to be bothersome.

`RingQQ` is the `FractionField` (Sec.32) of `RingZZ` (Sec.79);

See `RingElem` (Sec.71) for operations on its elements.

If you wish to compute purely with rationals (without exploiting CoCoALib's `ring` (Sec.69)s) then see the documentation in `BigRat` (Sec.9).

76.1.1 Examples

- ex-RingQQ1.C

76.1.2 Constructors and pseudo-constructors

- `RingQQ()` – produces the CoCoA `ring` (Sec.69) which represents QQ. Calling `RingQQ()` several times will always produce the same unique ring in CoCoALib.

76.1.3 Query

Let `R` be a `ring` (Sec.69)

- `IsQQ(R)` – says whether `R` is actually `RingQQ()`

76.1.4 Operations on RingQQ

See `FractionField` operations (Sec.32).

76.1.5 Homomorphisms

Let `S` be a `ring` (Sec.69)

- `NewQQEmbeddingHom(S)` – creates the (partial) homomorphism QQ –> `S` (but see also `CanonicalHom` (Sec.12)). QQ argument is implicit because there is a unique copy

76.2 Maintainer documentation for the class RingQQImpl

The function `RingQQ()` simply returns the unique instance of the CoCoALib `ring` (Sec.69) representing QQ. This instance is managed by `GlobalManager` (Sec.37), see its documentation.

The function `MakeUniqueInstanceOfRingQQ` is the only function which can call the ctor of `RingQQImpl`. The only function which is supposed to call `MakeUniqueInstanceOfRingQQ` is the ctor of `GlobalManager` (Sec.37). I have discouraged others from calling `MakeUniqueInstanceOfRingQQ` by not putting it in the header file `RingQQ.H` – see bugs section in `GlobalManager` (Sec.37).

`RingQQImpl` is the implementation of the field of rational numbers following the scheme laid by `RingBase` and `FractionFieldBase`. Almost all member functions are trivial: indeed, virtually all the work is done by the GMP library. Once you have understood how `RingZZImpl` works, the implementation here should be easy to follow.

The implementation of `RingQQImpl::InducedHomImpl::myApply` turns out to be a bit lengthy, but I do not see how to improve it. Since partial homomorphisms can be built, `myApply` maps numerator and denominator then must check that their images can be divided. I cannot reuse the implementation of `FractionFieldImpl::InducedHomImpl::myApply` because there is no equivalent of `RefNum` and `RefDen` in `RingQQImpl`.

76.3 Bugs, Shortcomings and other ideas

This code is probably not *exception safe*; I do not know what the `mpq_*` functions do when there is insufficient memory to proceed. Making the code "exception safe" could well be non-trivial: I suspect a sort of `auto_ptr` to an `mpq_t` value might be needed.

How to check that induced homomorphisms are vaguely sensible?? e.g. given ZZ->ZZ[x] ker=0, but cannot induce QQ->ZZ[x]; so it is not sufficient simply to check that the kernel is zero.

77 RingTwinFloat (John Abbott, Anna M. Bigatti)

77.1 User documentation for the classes RingTwinFloat and RingTwinFloatImpl

IMPORTANT NOTICE: please make sure you are using GMP 4.1.4 or later (wrong results may be obtained with earlier versions).

Elements of a `RingTwinFloat` try to act as though they were unlimited precision floating point values (while using only a finite precision). RingTwinFloat uses a heuristic to monitor loss of precision during computation, and will throw a `RingTwinFloat::InsufficientPrecision` object if it detects an unacceptable loss of precision. Beware that this is only a probabilistic heuristic which can underestimate precision loss. A `RingTwinFloat::InsufficientPrecision` object may also be caught as an `ErrorInfo` object having error code `ERR::InsuffPrec` (see `error` (Sec.25)).

EXAMPLE: If epsilon is a non-zero `RingTwinFloat` value then (1+epsilon == 1) will either be false or throw `RingTwinFloat::InsufficientPrecision`.

`RingTwinFloat` uses a heuristic for guessing when the difference of two almost equal values should be regarded as zero. While the heuristic is usually very reliable, it is possible to construct examples where the heuristic fails: see EXAMPLES/ex-RingTwinFloat1.C.

The function `IsInteger` will return false for any value of magnitude greater than or equal to 2^PrecisionBits(RR). Recognition of integers is heuristic; failures in ether sense are possible but are also unlikely.

See `RingElem` RingTwinFloat (Sec.71) for operations on its elements.

77.1.1 Examples

- ex-RingTwinFloat1.C
- ex-RingTwinFloat2.C

- ex-RingTwinFloat3.C

77.1.2 Pseudo-constructors

The constructor for RingTwinFloat takes a single argument being a lower bound on the number of bits' precision desired (in the mantissa). The value specified is probably rounded up a bit; exactly what happens depends on the mpf implementation in the GMP library. A minimum precision of 32 bits is imposed; smaller precisions are automatically increased to 32.

All arguments are `MachineInt` (Sec.46)

- `NewRingTwinFloat(AccuracyBits)`
- `NewRingTwinFloat(AccuracyBits, BufferBits, NoiseBits)`

77.1.3 Query and cast

Let `S` be a `ring` (Sec.69)

- `IsRingTwinFloat(S)` – `true` iff `S` is actually a `RingTwinFloat`
- `AsRingTwinFloat(S)` – if `S` is a `RingTwinFloat` *view* it as such

77.1.4 Operations

In addition to the standard `ring` operations (Sec.69), a `FractionField` may be used in:

- `PrecisionBits(RR)` – gives the mantissa precision specified in the ctor

77.1.5 Homomorphisms

Let `RR` be a `RingTwinFloat` and `R` any `Ring` (Sec.??)

- `NewApproxHom(RR, R)` – creates the homomorphism `RR -> S` (but see also `CanonicalHom` (Sec.12))

77.2 Maintainer documentation for the classes RingTwinFloat and RingTwinFloatImpl

As usual the class `RingTwinFloat` is just a reference counting smart pointer to an object of type `RingTwinFloatImpl` (which is the one which really does the work). The implementation of the smart pointer class `RingTwinFloat` is altogether straightfoward (just the same as any of the other smart pointer ring classes).

77.2.1 Philosophy

The implementation is based on Traverso's idea of "paired floats": each value is represented as two almost equal floating point numbers. The difference between the two numbers is intended to give a good indication of how much "noise" there is in the values. Here we shall allow larger tuples of floating point numbers. Arithmetic is performed independently on each component: e.g.

```
(a[0],a[1]) + (b[0],b[1])  ==>  (a[0]+b[0] , a[1]+b[1])
```

The consistency of the components is checked after every operation.

The main "trick" in the implementation of `RingTwinFloatImpl` is that its elements are `MultipleFloat`s (just a C array of `mpf_t` values). The number of components in a `MultipleFloat` value is determined by `RingTwinFloatImpl::myNumCompts` – currently fixed equal to 2 at compile time. Furthermore the values of these components must all be very close to each other. Indeed the function `RingTwinFloatImpl::myCheckConsistency` checks this condition: two outcomes are possible: - (1) all the components are very close to each other; - (2) at least one component is quite far from another. - In case (1) nothing more happens. In case (2) it is evident that an accumulated loss of precision has become unacceptable, and this triggers an exception of type

`RingTwinFloat::InsufficientPrecision`. The addition and subtraction functions check explicitly for near cancellation, and force the result to be zero in such cases.

The bit precision parameter specified when creating a RingTwinFloat is used in the following way (with the underlying principle being that elements of `RingTwinFloat(N)` should have at least roughly N bits of reliable value).

The digits in the mantissa (of each component in a `MultipleFloat`) are conceptually divided into three regions:

```
A A A A...A A A  B B B B....B B B B  C C C....C C C
<-  N bits   ->  <- sqrt(N) bits ->  <- N/2 bits ->
```

The region A comprises as many bits as the precision requested, and may be regarded as being correct with high probability. The region B comprises "guard digits": these digits are NOT regarded as being correct, but regions A and B of all components must be equal. Finally, region C is for "noise", and may be different in different components.

When an integer is converted to a `MultipleFloat`, the component with index 0 takes on the closest possible value to the integer while the other component(s) have about sqrt(N) bits of uniform random "noise" added to them (the random value may be positive or negative).

Special action is taken if there is large drop in magnitude during an addition (or subtraction): if the magnitude drops by more than N+sqrt(N) bits then the answer is forced to be equal to zero. There is a remote chance of erroneously computing zero when two almost equal values are subtracted. It does not seem to be possible to avoid this using limited precision arithmetic.

Special action is taken if a "noisy" component happens to be too close to the value at index 0: in this case more random noise is added. This can happen, for instance, if a value is divided by itself.

77.2.2 RingTwinFloatImpl::myFloor

It took me a while to find a satisfactory definition for the member function `myFloor` (even though the final code is fairly simple).

I eventually settled on the following definition. If the argument satisfies the `IsInteger` predicate then the floor function must surely give precisely that integer. Otherwise the argument (call it X) is not an integer, and the floor of X, if it exists, will be that integer N which satisfies the two-part condition $N < X$ and $N+1 > X$. If there is no such integer N then the floor cannot be computed, and an `InsufficientPrecision` exception must be thrown. In fact, there is an obvious candidate for N, namely the floor of the first component of the internal representation of X (it would be trickier to use the floor of the second component). Clearly N can be no larger than this candidate, since otherwise the first part of the condition would fail; and if N were any smaller then the second part would fail.

77.3 Bugs, shortcomings and other ideas

The code is ugly.

The functions `perturb`, `ApproximatelyEqual` and `myCmp` do "wasteful" alloc/free of temporary `mpf_t` values. `myCmp` can be done better.

What about a function which finds a continued fraction approximant to a `RingTwinFloat` value? It seems hard to implement such a function "outside" `RingTwinFloatImpl` as `InsufficientPrecision` will be triggered long before ambiguity is encountered in the continued fraction.

`myIsInteger` needs to be rewritten more sensibly (using `mpf_ceil` or `mpf_floor` perhaps?)

How to print out floats when they appear as coeffs in a polynomial??? What are the "best" criteria for printing out a float so that it looks like an integer? Should the integer-like printout contain a decimal point to emphasise that the value may not be exact?

Is it really necessary to call `myCheckConsistency` after multiplication and division? The accumulated loss of precision must grow quite slowly. Yes, it is necessary: consider computing 1^1000000 (or any other high power).

What about COMPLEX floats???

When a `MultipleFloat` is duplicated should its components be perturbed?

AsMPF is an UGLY function: signature reveals too much about the impl!

`myNumCompts` could be chosen by the user at run-time; in which case it must become a per-instance data member (instead of static). I'd guess that 2, 3 or 4 would be the best compromise.

Could it be useful to allow precisions below 32 bits? The limit does seem to be somewhat arbitrary. Perhaps the number of noise bits should also be allowed to vary?

`RingTwinFloatImpl::myOutput`:

- the the number of digits printed could be determined by how closely the different components match – would this be useful or wise?
- the number of digits printed is related to the definition of `myCheckConsistency` (I'm a little uneasy about this invisible link)

Should there be a means of mapping an element of a high precision `RingTwinFloat` to a lower precision `RingTwinFloat` (without having to pass through an external representation, such as a rational number)?

It seems wasteful to use two `mpf_t` values to represent a single `RingTwinFloat` value. Would it not be better to keep the main value and an "epsilon" (held as a `double` and an `int` exponent? Would it matter that "epsilon" has only limited precision?

77.4 Main changes

78 RingWeyl (John Abbott and Anna M. Bigatti)

78.1 User documentation

The class RingWeylImpl implements a Weyl algebra.

Note that Weyl algebras are noncommutative.

78.1.1 Examples

- ex-RingWeyl1.C
- ex-RingWeyl2.C
- ex-RingWeyl3.C
- ex-RingWeyl4.C
- ex-RingWeyl5.C

78.1.2 Constructors

- `NewWeylAlgebra(CoeffRing, NumTrueIndets, ElimIndets)`
- `NewWeylAlgebra(CoeffRing, names, ElimIndets)`

78.2 Maintainer documentation

This first version implements the Weyl algebra by using a normal polynomial ring internally (`myReprRing`) for manipulating the elements, and simply doing the right thing for products (instead of passing them directly onto `myReprRing`).

78.3 Bugs, shortcomings and other ideas

This documentation is extremely incomplete (time and energy are running out).

This version was produced in a considerable hurry, and worked by miracle.

There should be scope for some *optimization*, and perhaps some cleaning.

79 RingZZ (John Abbott, Anna M. Bigatti)

79.1 User documentation for RingZZ

The call `RingZZ()` produces the CoCoA `ring` (Sec.69) which represents ZZ, the ring of integers. Calling `RingZZ()` several times will always produce the same unique CoCoA `ring` (Sec.69) representing ZZ.

Strictly, there is a limit on the size of elements you can create, but the limit is typically high enough not to be bothersome.

See `RingElem` (Sec.71) for operations on its elements.

Efficiency of arithmetic on elements of `RingZZ()` should be reasonable rather than spectacular. If you wish to compute purely with integers (without exploiting CoCoALib's rings) then see the documentation in `BigInt` (Sec.8).

79.1.1 Examples

- ex-RingZZ1.C

79.1.2 Constructors and pseudo-constructors

- `RingZZ()` – produces the CoCoA `ring` (Sec.69) which represents ZZ. Calling `RingZZ()` several times will always produce the same unique ring in CoCoALib.

79.1.3 Query

Let `R` be a `ring` (Sec.69)

- `IsZZ(R)` – says whether `R` is actually `RingZZ()`

79.1.4 Homomorphisms

Let `S` be a `ring` (Sec.69)

- `NewZZEmbeddingHom(S)` – creates the homomorphism ZZ –> `S` (but see also `CanonicalHom` (Sec.12)). ZZ argument is implicit because there is a unique copy

79.2 Maintainer documentation for the class RingZZImpl

The function `RingZZ()` simply returns the unique instance of the CoCoALib `ring` (Sec.69) representing ZZ. This instance is managed by `GlobalManager` (Sec.37), see its documentation.

The function `MakeUniqueInstanceOfRingZZ` is the only function which can call the ctor of `RingZZImpl`. The only function which is supposed to call `MakeUniqueInstanceOfRingZZ` is the ctor of `GlobalManager` (Sec.37). I have discouraged others from calling `MakeUniqueInstanceOfRingZZ` by not putting it in the header file `RingZZ.H` – see bugs section in `GlobalManager` (Sec.37).

The class `RingZZImpl` is really very simple. It may look daunting and complex because it inherits lots of virtual functions from `RingBase`. It contains just three data members: a `MemPool` for managing the storage of the `mpz_t` headers, and pointers to the ring's own zero and one elements.

The member functions for arithmetic are all quite simple. The only minor difficulty is in the function `AsMPZ` which gets at the `mpz_t` hidden inside a `RingElemRawPtr`. I have decided to stick with the C interface to GMP for the moment (even though GMP 4 does offer a C++ interface). This appears to be more a personal choice than a technical one.

Recall (from `ring` (Sec.69)) that arithmetic on ring elements always passes via the virtual member functions of the concrete rings, and that these expect arguments to be of type `RawPtr` or `ConstRawPtr`. The arguments are pointers to the `mpz_t` headers which reside in a region of memory controlled by the `MemPool` (Sec.52) belonging to the `RingZZImpl` class.

Given that the `mpz_t` values must live on the free store, we use a `MemPool` (Sec.52) to handle the space for their headers (which are of fixed size). Note that this `MemPool` (Sec.52) is NOT what handles the memory used for the digits (or limbs) of the GMP integer values! Currently limb space is handled by whatever is the default allocator (`malloc`, I suppose).

The data members `myZeroPtr` and `myOnePtr` just hold `auto_ptr`s to the zero and one elements of the `RingZZImpl`. I used an `auto_ptr` to avoid having to worry about freeing it in the destructor; the zero and one values cannot be `RingElem`s because their creation must be deferred. I opted not to store the values in `RingElem` fields to avoid any possible problem due to a "race condition" where elements of the ring would be constructed before the body of the constructor of the ring had begun execution (might be OK anyway, but could easily lead to hair-raising bugs (*e.g.* in the dtor)).

79.3 Bugs, Shortcomings and other ideas

This code is probably not *exception safe*; I do not know what the `mpz_*` functions do when there is insufficient memory to proceed. Making the code "exception safe" could well be non-trivial: I suspect a sort of `auto_ptr` to an `mpz_t` value might be needed.

Should I switch to the C++ interface for GMP integers?

It is a shame that the `mpz_t` headers are "out of line". How much this may affect run-time performance I don't know.

Generation of random elements in `RingZZ` is not possible (yet???).

80 ServerOp (Anna Bigatti)

80.1 User documentation

80.1.1 Outline

`ServerOpBase` is the **abstract class** for an object representing an operation of the CoCoAServer. A concrete class must implement these functions (see below for a detailed description):

```
ServerOpBase(const LibraryInfo& lib)
void myOutputSelf(std::ostream&) const
void myReadArgs(std::istream& in)
void myCompute()
void myWriteResult(std::ostream&) const
void myClear()
```

The **concrete classes** representing the actual CoCoALib operations and their registrations are implemented in RegisterServerOps.C. See `RegisterServerOps` (Sec.68) for the registration procedure.

Data members

The class should have as data members the input `myIn..` and output variables `myOut..` for the main function called by `myCompute()`.

For example the class `IdealGBasis` has:

```
PolyList myInPL, myOutPL;
```

For data types **without a void constructor** use `auto_ptr`, for example the class `IdealElim` has:

```
auto_ptr<PPMonoidElem> myInElimPPPtr;
```

which is initialized in `IdealElim::myReadArgs`

```
myInElimPPPtr.reset(new PPMonoidElem(t));
```

LibraryInfo

A `LibraryInfo` is a set of information common to a group of operations. The **CoCoAServer** prints the list of loaded (sub)libraries at startup.

```
LibraryInfo(const std::string& name,
            const std::string& version,
            const std::string& group);
```

Example of definition of the function identifying a (sub)library:

```
// sublibrary of CoCoALib for groebner related operations
// by M.Caboara
const ServerOpBase::LibraryInfo& CoCoALib_groebner()
{
  static ServerOpBase::LibraryInfo UniqueValue("CoCoALib",
                                               BuildInfo::version,
                                               "groebner");
  return UniqueValue;
}
```

80.1.2 Virtual functions

myCompute

This function should be just a straight call to a CoCoALib function, in particular with neither reading nor printing, using as input the class members called `myIn..` and storing the result into the data members called `myOut..`, for example

```
void myCompute() { ComputeGBasis(myOutPL, myInPL); }
```

myReadArgs

Read from GlobalInput, and store the arguments into `myIn...` In general this is the only *difficult* function.

myWriteResult

Print the result(s) (myOut..) in CoCoA-4 language assigning it into the CoCoA4 global variable whose name is stored in `VarName4`. For *non-standard* output just remember it simply is CoCoA-4 language, for example:

```
void MVTN1::myWriteResult(std::ostream& out) const
{
  out << ourVarName4 << " := [];";
  for (unsigned int i=0; i<myOutPP.size(); ++i)
    out<< "Append(" << ourVarName4<< ", " << PP(myOutPP[i]) << ");" <<endl;
}
```

– add example for "Record[..];" output from ApproxBBasis –

myClear

Reset all data members to *0*. Right now (April 2007) it is only for *cleaning* the object right after it has been used, in future it might be called to *reuse* the object several times.

80.1.3 Debugging the server

If a function called by CoCoA-4 needs to be debugged this is the procedure to avoid dealing with sockets and fork under gdb.

- create from CoCoA-4 the input file `~/tmp/CoCoA4Request.cocoa5`:

  ```
  $cocoa5.Initialize();
  MEMORY.PKG.CoCoA5.PrintOnPath := GetEnv("HOME")+"/tmp";
  MyFun5(X);
  ```

- In shell:

```
src/server/CoCoAServer -d < ~/tmp/CoCoA4Request.cocoa5
```

- In gdb:

```
file src/server/CoCoAServer
r -d < ~/tmp/CoCoA4Request.cocoa5
break CoCoA::error
```

81 SmallFpDoubleImpl (John Abbott)

81.1 User documentation for SmallFpDoubleImpl

The class `SmallFpDoubleImpl` is a very low level implementation class for fast arithmetic in a small, prime finite field. It is **not intended** for use by casual CoCoALib users, who should instead see the documentation in `QuotientRing` (Sec.65) (in particular the function `NewZZmod`), or possibly the documentation in `RingFp` (Sec.72), `RingFpLog` (Sec.74), and `RingFpDouble` (Sec.73).

Compared to `SmallFpImpl` (Sec.82) the main difference is an implementation detail: values are represented as `double`s – on 32-bit computers this allows a potentially usefully greater range of characteristics at a probably minor run-time cost.

All operations on values must be effected by calling member functions of the `SmallFpDoubleImpl` class. Here is a brief summary.

```
SmallFpDoubleImpl::IsGoodCtorArg(p);   // true iff ctor SmallFpDoubleImpl(p) will succeed
SmallFpDoubleImpl::ourMaxModulus();    // largest permitted modulus
SmallFpDoubleImpl ModP(p, convention); // create SmallFpDoubleImpl object
long n;
BigInt N;
BigRat q;
SmallFpImpl::value_t a, b, c;

ModP.myModulus();         // value of p (as a long)

ModP.myReduce(n);         // reduce mod p
ModP.myReduce(N);         // reduce mod p
ModP.myReduce(q);         // reduce mod p

ModP.myExport(a);         // returns a preimage (of type long) according to symm/non-neg convention.

ModP.myNegate(a);         // -a mod p
ModP.myAdd(a, b);         // (a+b)%p;
ModP.mySub(a, b);         // (a-b)%p;
ModP.myMul(a, b);         // (a*b)%p;
ModP.myDiv(a, b);         // (a*inv(b))%p;  where inv(b) is inverse of b
ModP.myPower(a, n);       // (a^n)%p;  where ^ means "to the power of"
ModP.myIsZeroAddMul(a,b,c) // a = (a+b*c)%p; result is (a==0)
```

For `myExport` the choice between least non-negative and symmetric residues is determined by the convention specified when constructing the `SmallFpDoubleImpl` object. This convention may be either `GlobalSettings::SymmResidues` or `GlobalSettings::NonNegResidues`.

81.2 Maintainer documentation for SmallFpDoubleImpl

Most functions are implemented inline, and no sanity checks are performed (except when `CoCoA_DEBUG` is enabled). The constructor does do some checking. The basic idea is to use the extra precision available in `double`s to allow larger prime finite fields than are permitted when 32-bit integers are used for all arithmetic. If fast 64-bit

arithmetic becomes widespread then this class will probably become obsolete (unless you have a very fast floating point coprocessor?).

`SmallFpDoubleImpl::value_t` is simply `double`. Note that the values are always non-negative integers with maximum value less than `myModulusValue`; *i.e.* each residue class is represented (internally) by its least non-negative member.

To avoid problems with overflow the constructor checks that all integers from 0 to p*p-p can be represented exactly. We need to allow numbers as big as p*p-p so that `myIsZeroAddMul` can be implemented easily.

It is not strictly necessary that `myModulusValue` be prime, though division becomes only a partial map if `myModulusValue` is composite. I believe it is safest to insist that `myModulusValue` be prime.

81.3 Bugs, Shortcomings, and other ideas

The implementation is simplistic – I wanted to dash it off quickly before going on holiday :-)

82 SmallFpImpl (John Abbott)

82.1 User documentation for SmallFpImpl

The class `SmallFpImpl` is a very low level implementation class for fast arithmetic in a small, prime finite field. It is **not intended** for use by casual CoCoALib users, who should instead see the documentation in `QuotientRing` (Sec.65) (in particular the function `NewZZmod`), or possibly the documentation in `RingFp` (Sec.72), `RingFpLog` (Sec.74), and `RingFpDouble` (Sec.73).

The class `SmallFpImpl` offers the possibility of highly efficient arithmetic in small prime finite fields. This efficiency comes at a cost: the interface is rather unnatural and intolerant of mistakes. The emphasis is unequivocally on speed rather than safety or convenience.

The full speed of `SmallFpImpl` depends on many of its functions being inlined. The values to be manipulated must be of type `SmallFpImpl::value_t`. This is an unsigned machine integer type, and the values `0` and `1` may be used normally (but other values **must** be reduced before being used).

All operations on values must be effected by calling member functions of the `SmallFpImpl` class. Here is a brief summary.

```
SmallFpImpl::IsGoodCtorArg(p);   // true iff ctor SmallFpImpl(p) will succeed
SmallFpImpl::ourMaxModulus();    // largest permitted modulus
SmallFpImpl ModP(p, convention); // create SmallFpImpl object
long n;
BigInt N;
BigRat q;
SmallFpImpl::value_t a, b, c;

ModP.myModulus();         // value of p (as a long)

ModP.myReduce(n);         // reduce mod p
ModP.myReduce(N);         // reduce mod p
ModP.myReduce(q);         // reduce mod p

ModP.myExport(a);         // returns a preimage (of type long) according to symm/non-neg convention.

ModP.myNegate(a);         // -a mod p
ModP.myAdd(a, b);         // (a+b)%p;
ModP.mySub(a, b);         // (a-b)%p;
ModP.myMul(a, b);         // (a*b)%p;
ModP.myDiv(a, b);         // (a*inv(b))%p;  where inv(b) is inverse of b
ModP.myPower(a, n);       // (a^n)%p;  where ^ means "to the power of"
ModP.myIsZeroAddMul(a,b,c) // a = (a+b*c)%p; result is (a==0)
```

For `myExport` the choice between least non-negative and symmetric residues is determined by the convention specified when constructing the `SmallFpImpl` object. This convention may be either `GlobalSettings::SymmResidues`

or GlobalSettings::NonNegResidues.

82.1.1 Advanced Use: delaying normalization in a loop

The normal mod p arithmetic operations listed above always produce a normalized result. In some loops it may be possible to compute several iterations before having to normalize the result. The following three functions help implement such a delayed normalization strategy.

```
ModP.myNormalize(a);     -- FULL normalization of a
ModP.myHalfNormalize(a); -- *fast*, PARTIAL normalization of a
ModP.myMaxIters();
```

The value of myMaxIters() is the largest number of unnormalized products (of normalized values) which may be added to a partially normalized value before risking overflow. The partial normalization operation is quick (at most a comparison and a subtraction). Naturally, the final result must be fully normalized. See example program ex-SmallFp1.C for a working implementation.

82.2 Maintainer documentation for SmallFpImpl

Most functions are implemented inline, and no sanity checks are performed (except when CoCoA_DEBUG is enabled). The constructor does do some checking.

SmallFpImpl::value_t **must** be an unsigned integral type; it is a typedef to a type specified in CoCoA/config.H – this should allow fairly easy platform-specific customization.

This code is valid only if the square of myModulus can be represented in a SmallFpImpl::value_t; the constructor checks this condition. Most functions do not require myModulus to be prime, though division becomes only a partial map if it is composite; and the function myIsDivisible is correct only if myModulus is prime. Currently the constructor rejects non-prime moduli.

The code assumes that each value modulo p is represented as the least non-negative residue (*i.e.* the values are represented as integers in the range 0 to p-1 inclusive). This decision is linked to the fact that SmallFpImpl::value_t is an unsigned type.

The constants myResidueUPBValue and myIterLimit are to allow efficient exploitation of non-reduced multiplication (*e.g.* when trying to compute an inner product modulo p). See example program ex-SmallFp1.C

The return type of NumBits is int even though the result is always non-negative – I do not like unsigned values.

82.3 Bugs, Shortcomings, and other ideas

Should there be a myIsMinusOne function?

83 SmallFpLogImpl (John Abbott)

83.1 User documentation for SmallFpLogImpl

The class SmallFpLogImpl is a very low level implementation class for fast arithmetic in a small, prime finite field. It is **not intended** for use by casual CoCoALib users, who should instead see the documentation in QuotientRing (Sec.65) (in particular the function NewZZmod), or possibly the documentation in RingFp (Sec.72), RingFpLog (Sec.74), and RingFpDouble (Sec.73).

Compared to SmallFpImpl (Sec.82) the only difference is an implementation detail: multiplication and division are achieved using discrete log tables – this may be fractionally faster on some processors.

Note that the cost of construction of a SmallFpLogImpl(p) object for larger primes may be quite considerable (linear in p), and the resulting object may occupy quite a lot of space (*e.g.* probably about 6*p bytes).

All operations on values must be effected by calling member functions of the SmallFpLogImpl class. Here is a brief summary.

```
SmallFpLogImpl::IsGoodCtorArg(p);   // true iff ctor SmallFpLogImpl(p) will succeed
SmallFpLogImpl::ourMaxModulus();    // largest permitted modulus
```

```
SmallFpLogImpl ModP(p, convention); // create SmallFpLogImpl object
long n;
BigInt N;
BigRat q;
SmallFpImpl::value_t a, b, c;

ModP.myModulus();         // value of p (as a long)

ModP.myReduce(n);         // reduce mod p
ModP.myReduce(N);         // reduce mod p
ModP.myReduce(q);         // reduce mod p

ModP.myExport(a);         // returns a preimage (of type long) according to symm/non-neg convention.

ModP.myNegate(a);         // -a mod p
ModP.myAdd(a, b);         // (a+b)%p;
ModP.mySub(a, b);         // (a-b)%p;
ModP.myMul(a, b);         // (a*b)%p;
ModP.myDiv(a, b);         // (a*inv(b))%p;  where inv(b) is inverse of b
ModP.myPower(a, n);       // (a^n)%p;  where ^ means "to the power of"
ModP.myIsZeroAddMul(a,b,c) // a = (a+b*c)%p; result is (a==0)
```

For `myExport` the choice between least non-negative and symmetric residues is determined by the convention specified when constructing the `SmallFpLogImpl` object. This convention may be either `GlobalSettings::SymmResidues` or `GlobalSettings::NonNegResidues`.

83.2 Maintainer documentation for SmallFpLogImpl

The only clever bit is the *economical* construction of the log/exp tables in the constructor where we exploit the fact that `myRoot` to the power (p-1)/2 must be equal to -1.

This implementation uses discrete log/exp tables to effect multiplication and division quickly. Note that the residues themselves (*i.e.* the values of the ring elements) are held as machine integers whose value is the least non-negative representative of the residue class (*i.e.* in the range 0 to p-1). In particular, although log tables are used, we do NOT use a *logarithmic representation* for the field elements.

The log/exp tables are stored in C++ vectors: aside from their construction during the `RingFpLogImpl` constructor, these vectors are never modified, and are used only for table look-up. The C++ vectors are resized in the body of the constructor to avoid large memory requests when overly large characteristics are supplied as argument.

Besides these tables `SmallFpLogImpl` also remembers the characteristic in `myModulus`; `myRoot` is the primitive root used to generate the log/exp tables.

The members `myResidueUPBValue` and `myIterLimit` and `myHalfNormalize` may be used for delayed normalization in loops: see the inner product example in `SmallFpImpl` (Sec.82).

As the code currently stands, the modulus must also be small enough that it can fit into an `FpTableElem` (an `unsigned short`), and that its square can fit into a `value_t`. Using `short`s in the tables gave slightly better runtime performance in our tests. Furthermore, to permit use of unnormalized products in some algorithms, twice the square of the characteristic must fit into a `value_t` (*i.e.* `myIterLimit` must be greater than zero). The constructor for a `RingFpLogImpl` checks the size restrictions on the characteristic.

Note that the log table has a slot with index 0 which is never written to nor read from. The exp table is double size so that multiplication can be achieved more easily: the highest slot which could ever be used is that with index 2p-3 (in division), but the constructor fills two extra slots (as this makes the code simpler/neater).

The only slick part of the implementation is the filling of the tables in the constructor, where some effort is made to avoid doing more reductions modulo p than necessary. Note that the primitive root is always calculated (potentially costly!); there is no memorized global table of primitive roots anywhere.

83.3 Bugs, Shortcomings and other ideas

It is not as fast as I hoped – perhaps cache effects?

84 SmartPtrIRC (John Abbott)

84.1 User documentation for files SmartPtrIRC

The name `SmartPtrIRC` stands for *Smart Pointer with Intrusive Reference Count*. The desired behaviour is achieved through two cooperating classes: `SmartPtrIRC` and `IntrusiveReferenceCount`. These classes exist to facilitate implementation of smart pointers with reference counting. The suggested use is as follows. Make your implementation class inherit `protected`-ly from `IntrusiveReferenceCount`, and in your implementation class declare the class `SmartPtrIRC<MyClass>` as a friend. You can now use the class `SmartPtrIRC<MyClass>` as a reference counting smart pointer to your class.

The template argument of the class `SmartPtrIRC` specifies the type of object pointed to; if you want the objects pointed at to be `const` then put the keyword "const" in the template argument like this `SmartPtrIRC<const MyClass>`. Creating a new `SmartPtrIRC` to a datum will increment its reference count; conversely, destroying the `SmartPtrIRC` decrements the ref count (and destroys the object pointed at if the ref count reaches zero, see `IntrusiveReferenceCount::myRefCountDec`). Five operations are available for `SmartPtrIRC` values:

let `SPtr` be a `SmartPtrIRC` value

- `SPtr.myRawPtr()` returns the equivalent raw pointer
- `SPtr.operator->()` returns the equivalent raw pointer
- `SPtr.mySwap(SPtr2)` swaps the raw pointers
- `SPtr1 == SPtr2` returns true iff the equivalent raw pointers are equal
- `SPtr1 != SPtr2` returns true iff the equivalent raw pointers are unequal

The class `IntrusiveReferenceCount` is intended to be used solely as a base class. Note the existence of `IntrusiveReferenceCount::myRefCountZero` which forces the reference count to be zero. For instance, this is used in ring implementations where the ring object contains some *circular* references to itself; after creating the circular references the ring constructor then resets the reference count to zero so that the ring is destroyed at the right moment. SEE BUGS SECTION.

IMPORTANT NOTE: it is highly advisable to have `myRefCountZero()` as the very last operation in every contructor of a class derived from `IntrusiveReferenceCount`, i.e. intended to be used with `SmartPtrIRC`.

84.2 Maintainer documentation for files SmartPtrIRC

The entire implementation is in the `.H` file: a template class, and another class with only inline member functions. Inlining is appropriate as the functions are extremely simple and we expect them to be called a very large number of times.

The implementation is quite straightforward with one important detail: the destructor of `IntrusiveReferenceCount` must be virtual because `myRefCountDec` does a *polymorphic delete* through a pointer to `IntrusiveReferenceCount` when the count drops to zero. The book by Sutter and Alexandrescu gives wrong advice (in article 50) about when to make destructors virtual!

The fn `mySwap` is a member fn because I couldn't figure out how to make it a normal (templated?) function. I also feared there might have been some problems with the template fn `std::swap`.

84.3 Bugs, Shortcomings and other ideas

Should `myRefCountZero` be eliminated? It is not strictly necessary (just call `myRefCountDec` after each operation which incremented the ref count. This is related to how rings create their zero and one elements (and possibly other elements which should *always exist*, e.g. indets in a poly ring).

Could ref count overflow? Perhaps size_t is always big enough to avoid overflow?

It may be possible to replace all this code with equivalent code from the BOOST library. But so far (Nov 2006) the `shared_ptr` implementation in BOOST is not documented, so presumably should not be used. As there is no documentation I have not verified the existence of a *set ref count to zero* function; I rather suspect that it does not exist.

85 SmartPtrIRCCOW (John Abbott, Anna Bigatti)

85.1 User documentation for files SmartPtrIRCCOW

The name `SmartPtrIRCCOW` stands for *Smart Pointer with Intrusive Reference Count and Copy-on-write.* (or *Lazy Copy*).

It is very similar to `SmartPtrIRC` (Sec.84), where two cooperating classes are `SmartPtrIRCCOW` and `IntrusiveReferenceCountC` but also allows assigning, copying, and modifying.

85.2 Maintainer documentation for files SmartPtrIRCCOW

The abstract class `IntrusiveReferenceCountCOWBase` inherits from `IntrusiveReferenceCount` (see documentation for `SmartPtrIRC` (Sec.84)), with an additional pure virtual function `myClone` which must be implemented by the concrete class returning a deep copy of the object.

The template class `SmartPtrIRCCOW<T>` is implemented with one data member:

```
private:     SmartPtrIRC<T> mySmartPtr;
```

Which does (almost) all the work. The core for the *copy-on-write* behaviour is the member function:

```
private:     void myDetach()
```

which (if necessary) makes a new deep copy with reference count 1 and decrements the reference count of the original object.

85.3 Bugs, Shortcomings and other ideas

85.4 Main changes

2010

- `0.9938` first version July 2010 (experimental)

86 SocketStream (John Abbott)

86.1 User Documentation for SocketStream

86.1.1 General description

A `SocketStream` is intended to be used for client-server socket connections. The distinction between the two sorts of use is made explicit when the socket is created:

- the server end of a socket is created by specifying the port number on which to listen for connexions
- the client end of a socket is created by specifying both the machine name and port number to call

In both cases the `SocketStream` object is an `iostream`, *i.e.* it supports both input and output. Note that the constructor for a server end socket (*i.e.* one with just the port number as argument) will block until a connexion is received!

86.1.2 Example of Basic Use

Here is a simple, and rather silly, example. The server reads strings, and for each string read returns a string being the decimal representation of the length of the string received. Don't forget to start the server first, and then run the client (otherwise the client will complain about connexion refused).

86.1.3 Source for server.C

```
#include <string>
#include "CoCoA/SocketStreambuf.C"

int main()
{
  CoCoA::SocketStream s(8000); // server socket -- waits for a call

  while (s)
  {
    std::string str;
    s >> str;
    if (!s) break;
    std::cout << "Read the string: " << str << std::endl;
    s << str.size() << std::endl;
  }

  std::cout << "REACHED EOF -- QUITTING" << std::endl;
  return 0;
```

86.1.4 Source for client.C

```
#include <string>
#include <iostream>
#include "CoCoA/SocketStreambuf.C"

void process(const std::string& str, std::iostream& s)
{
  s << str << endl;
  std::string result;
  s >> result;
  std::cout << '"' << str << "\" transformed into \"" << result << '"' << std::endl;
}

int main()
{
  CoCoA::SocketStream s("point", 8000); // client socket

  process("String1", s);
  process("String2", s);
  process("archeopteryx", s);
  process("asuccessionofnonwhitespacecharacters", s);

  return 0;
}
```

86.2 Maintenance notes for the SocketStream source code

As mentioned below, most of this code was written by copying from other reliable sources – I don't really understand how it all works. For the `streambuf` code refer to Josuttis's excellent book. I do not know any formal reference for the "low-level" C code which uses the socket functions of the C library.

`SocketStreambuf::ourUngetSize` is a lower bound on how much one can "go backwards" using the `ungetc` function. `SocketStreambuf::ourInputBufferSize` is the size of the internal input byte buffer, so the maximum number of characters which can be read in a single call to "recv" is the difference between `ourInputBufferSize` and `ourUngetSize` (currently 99996 bytes).

The constructor for a server size `SocketStream` internally calls "fork" when a connexion is received – the constructor completes only in the child, the parent process waits for further connexions.

86.3 Bugs, Shortcomings, etc

I do not like having to include <cstdio> just to get the preprocessor macro EOF

ERROR HANDLING NEEDS TO BE RECONSIDERED. Error handling is probably not correct: too great a tendency to throw exceptions instead of simply putting the iostream into an "anomalous state". Not sure what is the accepted C++ approach.

The values for the constants SocketStreambuf::ourInputBufferSize and SocketStreambuf::ourUngetSize are rather arbitrary.

Most of the code has been "ripped off": either from Daniele's C source, or from Josuttis's book. I have felt free to make (wholesale) changes.

Maintainer documentation is largely absent.

87 SparsePolyRing (John Abbott)

87.1 Examples

- ex-PolyRing1.C
- ex-PolyRing2.C
- ex-PolyIterator1.C
- ex-PolyIterator2.C
- ex-PolyInput1.C
- ex-NF.C

87.2 User documentation for SparsePolyRing

SparsePolyRing is an abstract class (inheriting from PolyRing (Sec.63)) representing rings of polynomials; in particular, rings of sparse multivariate polynomials (e.g. written with *sparse representation*) with a special view towards computing Groebner bases and other related operations. This means that the operations offered by a SparsePolyRing on its own values are strongly oriented towards those needed by Buchberger's algorithm.

A polynomial is viewed abstractly as a formal sum of ordered terms; each term is a formal product of a non-zero coefficient (belonging to the coefficient ring (Sec.69)), and a power product of indeterminates (belonging to the PPMonoid (Sec.58) of the polynomial ring). The ordering is determined by the PPOrdering (Sec.60) on the power products: distinct terms in a polynomial must have distinct power products. The zero polynomial is conceptually the formal sum of no terms; all other polynomials have a *leading term* being the one with the largest power product (PPMonoidElem (Sec.??)) in the given ordering.

See RingElem SparsePolyRing (Sec.71) for operations on its elements.

87.2.1 Pseudo-constructors

Currently there are four functions to create a polynomial ring:

NewPolyRing(CoeffRing, NumIndets) This creates a sparse polynomial ring with coefficients in CoeffRing and having NumIndets indeterminates. The PP ordering is StdDegRevLex. CoCoALib chooses automatically some names for the indeterminates (currently the names are x[0], x[1], ... , x[NumIndets-1]).

NewPolyRing(CoeffRing, IndetNames) This creates a sparse polynomial ring with coefficients in CoeffRing and having indeterminates whose names are given in IndetNames (which is of type vector<symbol>). The PP ordering is StdDegRevLex.

NewPolyRing(CoeffRing, IndetNames, ord) This creates a sparse polynomial ring with coefficients in CoeffRing and having indeterminates whose names are given in IndetNames (which is of type vector<symbol>). The PP ordering is given by ord.

NewPolyRing(CoeffRing, PPM) This creates a sparse polynomial ring with coefficients in CoeffRing and with power products in PPM which is a power product monoid which specifies how many indeterminates, their names, and the ordering on them.

87.2.2 Query and cast

Let `R` be an object of type `ring` (Sec.69).

- `IsSparsePolyRing(R)` – `true` if `R` is actually `SparsePolyRing`
- `AsSparsePolyRing(R)` – if `R` is a `SparsePolyRing` *view* it as such

87.2.3 Operations on a SparsePolyRing

In addition to the standard `PolyRing` operations (Sec.63), a `SparsePolyRing` may be used in other functions.

Let `P` be an object of type `SparsePolyRing`.

- `PPM(P)` – the PPMonoid of `P`.
- `GradingDim(P)` – the dimension of the grading on `P` (may be 0).

87.2.4 Operations with SparsePolyIters

A `SparsePolyIter` (class defined in SparsePolyRing.H) is a way to iterate through the summands in the polynomial without knowing the (private) details of the concrete implementation currently in use. (The idea is similar to C++ iterators for STL containers.)

Let `f` denote a non-const element of P. Let `it1` and `it2` be two `SparsePolyIter`s running over the same polynomial.

- `BeginIter(f)` – a `SparsePolyIter` pointing to the first term in `f`.
- `EndIter(f)` – a `SparsePolyIter` pointing to one-past-the-last term in `f`.

  Changing the value of `f` invalidates all iterators over `f`.
- `coeff(it1)` – read-only access to the coeff of the current term
- `PP(it1)` – read-only access to the pp of the current term
- `++it1` – advance `it1` to next term, return new value of `it1`
- `it1++` – advance `it1` to next term, return copy of old value of `it1`
- `it1 == it2` – true iff `it1` and `it2` point to the same term; throws `CoCoA::ErrorInfo` with code `ERR::MixedPolyIters` if `it1` and `it2` are over different polys.
- `it1 != it2` – same as `!(it1 == it2)`
- `IsEnded(it1)` – true iff `it1` is pointing at the one-past-the-last term

Examples

- ex-PolyIterator1.C
- ex-PolyIterator2.C

87.3 Maintainer documentation for SparsePolyRing

The exact nature of a *term* in a polynomial is hidden from public view: it is not possible to get at any term in a polynomial by any publicly accessible function. This allows wider scope for trying different implementations of polynomials where the *terms* may be represented in some implicit manner. On the other hand, there are many cases where an algorithm needs to iterate over the terms in a polynomial; some of these algorithms are *inside* PolyRing (i.e. the abstract class offers a suitable interface), but many will have to be *outside* for reasons of modularity and maintainability. Hence the need to have *iterators* which run through the terms in a polynomial.

The implementations in SparsePolyRing.C are all very simple: they just conduct some sanity checks on the function arguments before passing them to the PolyRing member function which will actually do the work.

87.4 Bugs, Shortcomings and other ideas

Too many of the iterator functions are inline. Make them out of line, then use profiler to decide which should be inline.

`PushFront` and `PushBack` do not verify that the ordering criteria are satisfied.

Verify the true need for `myContent`, `myRemoveBigContent`, `myMulByCoeff`, `myDivByCoeff`, `myMul` (by pp). If the coeff ring has zero divisors then `myMulByCoeff` could change the structure of the poly!

Verify the need for these member functions: myIsZeroAddLCs, myMoveLM, myDeleteLM, myDivLM, myCmpLPP, myAppendClear, myAddClear, myAddMul, myReductionStep, myReductionStepGCD, myDeriv.

Should there be a RingHom accepting IndetImage (in case of univariate polys)?

88 submodule (John Abbott, Anna M. Bigatti)

88.1 Examples

- ex-module2.C

88.2 User documentation

Here are some pseudo-constructors for modules which are generated by a (finite) vector of `ModuleElem` (Sec.**??**) (in a `FreeModule` (Sec.33)). There is no class for submodules, they are objects of type `FGModule` (Sec.30).

There are several ways to create a submodule:

- `submodule(M, gens)` – creates an `FGModule` (Sec.30) representing the submodule of the FGModule M generated by the elements in gens; note that M must be specified even though it is usually implicit in the values contained in gens (unless gens is empty).
- `SubmoduleCols(M, A)` – the submodule generated by the columns of the `matrix` (Sec.47) `A`.
- `SubmoduleRows(M, A)` – the submodule generated by the rows of the `matrix` (Sec.47) `A`.

88.2.1 Operations

The permitted operations on submodules are:

```
FGModule SyzOfGens(const FreeModule& F, const ideal& I);
FGModule SyzOfGens(const FreeModule& F, const FGModule& N);
FGModule SyzOfGens(const ideal& I);
FGModule SyzOfGens(const FGModule& N);
bool IsElem(const ModuleElem& v, const module& M);
bool IsContained(const module& M, const module& M);
```

88.3 Maintainer documentation for the classes module, and ModuleElem

I shall suppose that the maintainer documentation for modules and FGModules has already been read and digested. It could also be helpful to have read ring.txt since the "design philosophy" here imitates that used for rings.

SubmoduleImpl is a concrete class derived from FGModuleBase, i.e. objects of this class represent submodules of explicitly finitely generated modules. The data members comprise the two obvious values:

```
FreeModule myM;                 // the ambient module in which the generators live
vector<ModuleElem> myGensArray; // the generators as specified by the user
```

Additionally there are two other data members:

```
bool myTidyGensIsValid;             // true iff myTidyGensArray contains a correct value
vector<ModuleElem> myTidyGensArray; // a "nice" set of generators
```

It is difficult to be precise about the value which myTidyGensArray should contain (when valid) since it depends upon the module. If the module is over a polynomial ring then it will be a Groebner basis. If the module is over Z then it will presumably be either a "Hermite Basis" or an "LLL Basis".

88.4 Bugs, Shortcomings and other ideas

Implementation and documentation are rather incomplete.

Why is myM a FreeModule and not an FGModule???

What is myTidyGensArray for a module over Z???

89 SugarDegree (Anna Bigatti)

89.1 User documentation

Abstract class for implementing several kinds of *sugar*:

- homogenous case (= the degree)
- non graded case (using StdDeg)
- graded case (using wdeg)
- non graded case and saturating algorithm
- graded case and saturating algorithm
  === Pseudo constructors ===
- `NewStdSugar(ConstRefRingElem f);`
- `NewStdSugarNoIdx(ConstRefRingElem f, long PosIndet);`
- `NewStdSugarSat(ConstRefRingElem f);`
- `NewStdSugarNoIdxSat(ConstRefRingElem f, long PosIndet);`
- `NewWSugar(ConstRefRingElem f);`
- `NewWDeg1CompTmp(ConstRefRingElem f);` – temporary: only for testing
- `NewWSugarConst(ConstRefRingElem f);` – stays constant in myUpdate
- `NewWSugarSat(ConstRefRingElem f);`
  There is also an "empty" constructor for when you don't have yet enough information to choose the kind of sugar. However it does require the `uninitialized` marker to make sure you know you have an uninitialized sugar!
- `sugar(uninitialized);`

89.1.1 Member functions

Warning! The following throw an error if the wrong type of value is asked!

- `const degree& myWSugar() const =0;` – only if impl stores this value
- `long myStdSugar() const =0;` – only if impl stores this value
  Warning! The following throw an error if the sugar is not initializes!
- `void myMul(ConstRefPPMonoidElem pp) =0;` – sugar after multiplying by pp
- `void myUpdate(ReductionCog F, const GPoly& g);` – sugar after reducing F by g
- `void myUpdate(ConstRefPPMonoidElem CofactorPP, const GPoly& g) =0;` – sugar after adding pp*g
- `int myCmp(const SugarDegreeBase& s) const =0;` – this <=> s ? <0,=0,>0
- `std::ostream& myOutput(std::ostream& out) const =0;`

89.1.2 Non member functions

- `bool IsInitialized(const SugarDegree& sd);`
- `std::ostream& operator<<(std::ostream& out, const SugarDegree& s);`

  == Maintainer documentation ==

  Work in progress

  Sugar has not been properly tested on modules

  == Bugs, shortcomings and other ideas ==

90 symbol (John Abbott)

90.1 Examples

- ex-symbol1.C
- ex-symbol2.C
- ex-PPMonoidElem1.C
- ex-PolyRing2.C

90.2 User documentation

`symbol` is short for *Symbolic Name*. A value of type `symbol` represents a *variable name* possibly with some integer subscripts attached. Its primary use is for input and output of polynomials: the name of each indeterminate in a `PolyRing` (Sec.63) is a `symbol`, similarly for a `PPMonoid` (Sec.58).

A `symbol` value has two components:

- **head** – a string starting with a letter followed by letters and _s (**note** no digits or other characters allowed)
- **subscripts** – a vector of machine integers, possibly empty (subscripts may be negative)

Examples of `symbol`s are: (in standard printed forms)

`x`, `X`, `alpha`, `z_alpha`, `x[2]`, `gamma[-2,3,-9]`

90.2.1 Anonymous symbols

It is also possible to create anonymous symbols: they are used for building *temporary* polynomial extensions on unknown coefficient rings (which may contain any symbol).

- **head** – is the *empty* string
- **subscripts** – exactly one subscript

Each newly created anonymous symbol has a subscript strictly greater than that of any previous anonymous symbol. For better readability, an anonymous symbol prints out as a **hash** followed by the subscript: *e.g.* `#[12]`

90.2.2 Constructors

Let `head` be a `std::string`, `ind`, `ind1`, `ind2`, `n` machine integers, `inds` a `std::vector<long>`.

- `symbol(head)` – a `symbol` with no subscripts
- `symbol(head, ind)` – a `symbol` with a single subscript
- `symbol(head, ind1, ind2)` – a `symbol` with a two subscripts
- `symbol(head, inds)` – a `symbol` with the given subscripts
- `NewSymbol()` – a new anonymous symbol (prints as, for example, `#[12]`)

Creating a vector of symbols

Several polynomial ring pseudo-constructors expect a `vector` of `symbol`s to specify the names of the indeterminates. There are several convenience functions for constructing commonly used collections of `symbol`s.

- `symbols(hd1)` – create vector of length 1 containing `symbol(hd1)`
- `symbols(hd1,hd2)` – ... length 2...
- `symbols(hd1,hd2,hd3)` – ... length 3...
- `symbols(hd1,hd2,hd3,hd4)` – ... length 4...
- `SymbolRange(hd, lo, hi)` – create vector of `hd[lo]`, `hd[lo+1]`, ... `hd[hi]`. Note that these symbols each have just a single subscript
- `SymbolRange(sym1, sym2)` – create vector of *cartesian product* of the subscripts, e.g. given `x[1,3]` and `x[2,4]` produces `x[1,3]`, `x[1,4]`, `x[2,3]`, `x[2,4]`
- `NewSymbols(n)` – `n` new anonymous symbols

90.2.3 Operations on symbols

Let `sym`, `sym1`, and `sym2` be objects of type `symbol`

- `head(sym)` – head of `sym` as a const ref to `std::string`
- `NumSubscripts(sym)` – number of subscripts `sym` has (0 if sym has no subscripts)
- `subscript(sym, n)` – gives `n`-th subscript of `sym`
- `cmp(sym1, sym2)` – <0, $=0$, >0 according as `sym1` $<$ $=$ $>$ `sym2` (for more info see Maintainer section)
- `sym1` $<$ `sym2` – comparisons defined in terms of `cmp`
- `sym1` $<=$ `sym2` – ...
- `sym1` $>$ `sym2` – ...
- `sym1` $>=$ `sym2` – ...
- `sym1 == sym2` – ...
- `sym1 != sym2` – ...
- `out` $<<$ `sym` – print `sym` on `out`
- `in` $>>$ `sym` – read a symbol into `sym` (but also see Bugs section) (expected format is x, y[1], z[2,3], etc.)

Operations on vectors of symbols

- `AreDistinct(vecsyms)` – true iff all symbols are distinct
- `AreArityConsistent(vecsyms)` – true iff all symbols with the same head have the same arity

90.3 Maintainer documentation for symbol

Note: I have used `MachineInt` as the type for fn args containing index values because it is *safer*, and I believe that the run-time penalty is unimportant. This is a considered exception to the guideline which says to use `long` for indexes.

I have decided **not** to allow *big integers* as subscripts because I don't see when it could ever be genuinely useful.

The implementation is extremely simple. Efficiency does not seem to be important (*e.g.* `symbols` and `SymbolRange` copy the vector upon returning). The implementation of `SymbolRange` is mildly delicate when we have to make checks to avoid integer overflow – see comments in the code.

To make "anonymous" symbols I opted to use a **private ctor** which accepts just a single subscript; this ctor is called only by `NewSymbol` and `NewSymbols`.

The printing fn (`myOutputSelf`) has to check for an empty head, and if found it prints the string in `AnonHead`.

We believe a total ordering on `symbol`s could be useful; for instance, if someone wants to make a `std::map` using `symbol`s. Currently the total order is *Lex on the heads then lex on the subscript vectors*; this is simple, and is probably fast enough.

The function `symbol::myInput` is a stop-gap implementation.

90.4 Bugs, Shortcomings and other ideas

The member function `myInput` handles white space wrongly. For CoCoALib whitespace is space, TAB, or backslash-newline; newline without backslash is not considered white space.

It might be nice to have a function which returns the vector of subscripts of a name.

I wonder what sending a `symbol` on an OpenMath channel would mean (given that OpenMath is supposed to preserve semantics, and a symbolic name is by definition devoid of semantics).

91 ThreadsafeCounter (John Abbott)

91.1 User documentation for ThreadsafeCounter

A `ThreadsafeCounter` is simply a counter (based on a `long`) which may be incremented in a threadsafe manner.

91.1.1 Constructors

There is only one constructor, the default constructor:

- `ThreadsafeCounter()` – create new counter starting at zero.

91.1.2 Operations on ThreadsafeCounters

There is only one operation:

- `TCS.myAdvance(n)` – increment the counter by `n`, and return the value of the counter prior to incrementing.

Note that `myAdvance` is likely to be quite slow as it uses mutexes.

A `ThreadsafeCounter` may be printed; this is intended primarily to permit debugging.

91.2 Maintainer documentation

I copied the code from a BOOST example. It's so simple there are obviously no deficiencies!

`operator<<` is probably not threadsafe (but does that matter?)

91.3 Bugs, shortcomings and other ideas

No check for overflow!

You cannot query the counter's value without incrementing it.

91.4 Main changes

2012

- 05-May (v0.9951): first version

92 time (John Abbott)

92.1 User documentation for CpuTime and RealTime

`CpuTime()` returns a `double` whose value is the user CPU usage in seconds since the start of the program (*i.e.* the amount of time the processor has dedicated to your computation – this may be rather less than the real elapsed time if the computer is also busy with other tasks). For instance, to find out how long `func()` takes to execute you can do the following:

```
int main()
{
  double t0 = CpuTime();
  func();
  cout << "Time taken (in seconds) is " << CpuTime()-t0 << endl;
  return 0;
}
```

In contrast the function `RealTime()` returns a `double` whose value is the number of seconds elapsed since some fixed point in the past (on Unix/Linux boxes this is typically 1st January 1970, sometimes called "the epoch").

WARNING we cannot guarantee the accuracy of these functions; as a rule of thumb you should regard time differences as having an imprecision of around 2% plus upto 0.2 seconds of unknown variation. So using these functions to measure a time difference less than 1 second is likely to produce a value with quite a large relative error.

As a convenience there is also the function `DateTime(long& date, long& time)` which stores in `date` and `time` the current date and time represented as decimal integers having the formats `yyyymmdd` & `hhmmss` respectively. Example:

```
long date, time_unused;
DateTime(date, time_unused);
int YearToday = date/10000;
int MonthToday = (date/100)%100;
int DayToday = date%100;
```

92.2 Maintainer documentation for CpuTime

It works on GNU/Linux and MacOSX. I hope someone else will deal with the portability issues.

92.3 Bugs, Shortcomings, and other ideas

Might not work on Microsoft platforms – maybe this is really a feature?

I ignore the return values of `getrusage` and `gettimeofday`; I'd be amazed if they could signal errors, but perhaps the code ought to check?

BOOST has probably solved this; apparently Bruno has a solution too.

93 ULong2Long (John Abbott)

93.1 User documentation

93.1.1 Generalities

The function `ULong2Long` converts an `unsigned long` value into a `signed long`, effectively inverting the standard C++ cast from `signed long` to `unsigned long`. Note that applying a `static_cast` might not produce the desired result – officially the outcome is "implementation defined".

93.2 Maintainer Documentation

There are three different implementations. The choice between them is determined by the value of the CPP symbol `COCOA_ULONG2LONG`; a suitable value for this symbol is found by a script called by the `configure` script. That script

selects the simplest implementation which works (on certain test cases). Note that C++ explicitly forbids the use of `reinterpret_cast` on built-in integral types, but the trick of applying to a reference seems to work (it was suggested to me by Chris Jefferson).

An earlier version of this function was in `utils.H`, but it turned out to be simpler to place it by itself in a separate header file (because the `ULong2Long.H` includes no further headers, so the test compilations made by the script `cpp-flags-ulong2long.sh` are simpler and safer).

Everything is in the header file; there is no `ULong2Long.C` file.

93.3 Bugs, shortcomings and other ideas

The fully portable definition is long and slow – this seems to be a problem of the C++ standard.

93.4 Main changes

2011

- August (v0.9950):
  - first robust version (with configure-time selection)

94 utils (John Abbott)

94.1 User documentation for file utils.H

This file defines a few very basic functions which I feel should really be part of the standard C++. Nevertheless I have placed all definitions inside the namespace `CoCoA`. Here is a summary:

- the `DeleteObject` struct is useful when using the C++ standard library containers to hold plain pointers to data they own. I took it from Scott Meyers's book "More Effective STL".
- a template function `cmp` which conducts a three-way comparison of its two arguments: cmp(a, b) returns -1 if a<b, returns 0 if a==b, and returns 1 if a>b – you can think of cmp(a,b) = sgn(a-b).
- a template function `LexCmp3` which conducts a three-way lex comparison of two sequences (specified by start/end iterators). The call LexCmp3(StartA, EndA, StartB, EndB) gives the result -1, 0, or 1 according as sequence A is lexicographically before sequence B, equal to sequence B, or after sequence B.

94.2 Maintainer documentation for files utils.H

Everything is in `utils.H`; the functions are all so simple that they can be implemented inline.

The type `int` seemed the most natural choice for the return value of the three-way comparison functions (though `signed char` would be big enough). The implementation assumes that `operator<` is defined; this decision was inspired by assumptions made by various STL functions. The types of the arguments may be different as this is probably be more convenient for the user. Obviously the generic definition given here can be overridden by more efficient specific definitions for certain argument types.

94.3 Bugs, Shortcomings and other ideas

Should the template function `cmp` require its args to be exactly the same type?