MuPAD-Combinat, an open-source package for research in algebraic combinatorics

Florent Hivert [*]   Nicolas M. Thiéry [†]

4th May 2004

Abstract

In this article we give an overview of the MuPAD-Combinat open-source algebraic combinatorics package for the computer algebra system MuPAD 2.0.0 and higher. This includes our motivations for developing yet another combinatorial software, a tutorial introduction with lots of examples, as well as notes on the general design. The material presented here is also available as a part of the MuPAD-Combinat handbook; further details and references on the algorithms used can be found there. The package and the handbook are available from the web page, together with download and installation instructions, mailing-lists, etc.

`http://mupad-combinat.sourceforge.net`

Dedicated to Alain Lascoux, on the occasion of his 60th birthday.

[*]Institut Gaspard Monge, Université de Marne-la-Vallée, 77454 Marne-la-Vallée Cedex 2, France; `Florent.Hivert@univ-mlv.fr`

[†]Laboratoire de Mathématiques Discrètes, Université Lyon I, 43 bd du 11 novembre 69622 Villeurbanne Cedex, France; `nthiery@users.sf.net`

Contents

1 Introduction

`MuPAD-Combinat` is an open-source algebraic combinatorics package for the computer algebra system MuPAD 2.0.0 and higher. The main purpose of this package is to provide an extensible toolbox for computer exploration. The development started in spring 2001, and the package currently contains functions to deal with usual combinatorial classes (partitions, tableaux, decomposable classes, ...), Schubert polynomials, characters of the symmetric group, and weighted automata. It supplies the user with tools for constructing new combinatorial classes and combinatorial algebras and, as an application, provides some well-known combinatorial algebras like the algebra of symmetric functions and various generalizations. Most of the code derives from computer exploration while doing research on the following topics: invariant theory of permutation groups [?, ?, ?], Steenrod algebras and symmetric functions [?], binary tree algebras such as the Loday-Ronco algebra and renormalization in quantum electrodynamics [?, ?], representation theory, quantum groups, and homological computations [?], as well as peaks and Hecke-Clifford Algebras [?]. This represents about 65000 lines of MuPAD and C++ code together with 450 pages of documentation, written by 3 main developers and altogether about 20 contributers. The core of the package is integrated in the official library of MuPAD since version 2.5.0.

The purpose of this paper is to present the package for novice and advanced users, for potential contributers, as well as for developers who do not know about MuPAD but want to compare the package with other similar packages.

After a presentation of our motivations for writing such a package, we propose a guided tour through its features. Though this tour assumes some familiarity with the MuPAD computer algebra system, the first part which describes the combinatorial feature of `MuPAD-Combinat` is intended for novice users, and does not assume strong programming knowledge. The second part of the tour which is devoted to the building of new combinatorial algebras, is a little bit more involved.

After that, the paper goes on with some design notes. This third part of the paper deals much more with programming techniques, and may be of interest for people wanting to understand the underlying mechanisms of the package, for example to compare it with similar packages. It assumes some strong knowledge about programming but not necessarily about the MuPAD language itself. In particular we discuss the advantage of using typing mechanism and object oriented features rather than just manipulating expressions which is the usual mechanism of similar packages. As such, it may interest people wanting to have comparable features in a different language.

1.1 A need for a toolbox for computer exploration in algebraic combinatorics

While doing research in (algebraic) combinatorics, computer exploration can be of great help. In its simplest form, when looking for the generating series of a combinatorial class, one can try to compute its first terms; those may give a hint on a recurrence relation or a general formula; at least, they can be sent to the *Online Encyclopedia of Integer Sequences* [?] for comparison with well known sequences. In general, using a computer allows one to study large scale examples (in combinatorics, the size of examples usually grows very quickly!). This can help to suggest conjectures, check them for likeliness, or find counter-examples.

The first author is interested in symmetric functions and their generalizations in connection with representation theory. The problem is basically to find interesting bases together with product and change of basis rules (analogues of Littlewood-Richardson rules). His results were mainly obtained by computer exploration, using some `Maple` routines extending the `ACE` package [?]. Similarly, the second author had developed a library for computing within invariant rings of permutation groups, in order to study certain invariant rings related to graph theory. The common point of those tools was that they essentially consisted of basic combinatorial routines together with mechanics to compute within certain combinatorial algebras.

By a combinatorial algebra we mean a vector space with a basis indexed by combinatorial objects and endowed with a product that obeys some combinatorial rule. Think of the group algebra of the n-th symmetric group: the basis is indexed by permutations, while the product is given by the usual product of permutations. Such combinatorial algebras appear in many situations (see e.g. [?, ?, ?, ?]). One often needs to run computations within such algebras, for example for finding generators, idempotents, and in general better understanding their algebraic structure.

A typical problem is to find the elements c in a given combinatorial algebra satisfying certain properties (say $c^2 = c$):

1. Provide the product rule on basis elements (unless the algebra is already implemented);
2. Produce a system of equations characterizing those elements c by running appropriate computations in the algebra;
3. Solve this system of equations;
4. Interpret the result.

This simple example highlights what a platform for computer exploration should essentially provide: in step 1, a comprehensive toolbox of basic combinatorial routines may help to implement the product rule; in step 2, the system should take care of all the linear bookkeeping to allow one to easily manipulate elements of the algebra; step 3 requires all the usual computer algebra tools (linear algebra, Gröbner bases, integration, solvers, ...).

1.2 Review of preexisting software

We now review some available software tools for doing algebraic combinatorics and we comment from our experience and expectations why or why not, or to what extent, they fit our needs. We do not seek completeness, but rather want to present the background that motivated the definition of the specifications for `MuPAD-Combinat`. For a comprehensive list of related software, we refer to `http://www.mat.univie.ac.at/~slc/divers/software.html`.

One of the first and widely known packages for algebraic combinatorics is J. Stembridge's `SF` library for `Maple` [?] which is designed to compute with symmetric functions.

A more ambitious package for `Maple` [?], called `ACE` [?], was developed in Marne-la-Vallée, mostly by S. Veigneau. It provides a wide range of combinatorial routines and implements several classical combinatorial algebras (symmetric functions, quasi-symmetric function, non commutative symmetric functions, Schubert polynomials, ...) using state of the art algorithms. Being a library for a computer algebra system that is widely used in the community helped it to spread (there are about 100 known users), and allowed one to combine it with the many other existing combinatorics package for the same system. On the other hand it suffered from the poor programming language of `Maple`, and the notorious incompatibilities with the new versions of `Maple` made its maintenance tricky. Experience showed that the overall design made it difficult to extend, in particular for defining new combinatorial algebras. Altogether, the development essentially stalled in 1999 when S. Veigneau left for industry after finishing his PhD thesis.

μ-`EC` [?], also developed in Marne-la-Vallée, by V. Prosper, was an attempt to translate `ACE` for the computer algebra system MuPAD. The goal was mainly to test whether the MuPAD programing language was more adapted to the needs, and in particular to incorporate `Symmetrica` [?] (see below) into the system, via a dynamic module. However, it suffered from the same design and development model limitations as `ACE` and the development also stalled when V. Prosper left for industry after finishing his PhD thesis in 2000.

A. Kohnert leads the development of `Symmetrica` [?], a collection of `C` routines to compute with symmetric functions and Schubert polynomials, ordinary, modular, and projective representations of the symmetric group, and Hecke algebras of type A. The underlying programming language permits very substantial speed improvements compared to equivalent algorithms written, say, in `Maple`. The object oriented design definitely helps for maintaining and extending it. On the other hand, it does not provide support for an easy definition of new combinatorial algebras, and can't be straightforwardly combined with other computer algebra tools. The remaining drawbacks, coming from the programming language, are partly matters of personal taste. There is a steep learning curve for casual programmers, which makes it difficult to attract new users. We also find the development cycle to be too long in a low-level programing language. Finally, not having an interpreter makes it quite unpractical for interactive computer exploration (this could be circumvented by using a `C` interpreter like `CINT`).

B. Weybourne also wrote an interactive program called `Schur` for calculating properties of Lie groups and symmetric functions, with a view toward physics. As for `Symmetrica`, it can't be easily combined with other computer algebra tools, and does not provide support for easy definition of new combinatorial algebras.

To some extent, the systems `GAP` [?] and `Magma` [?, ?] allow the user to define new combinatorial (Lie) algebras, and provide a wide set of tools from group theory and algebra that are useful for algebraic combinatorics. We discuss them further later on in the choice of the underlying system.

Finally, one should mention the `Maple` library for Gröbner basis computations by F. Chyzak which allows one to easily implement those combinatorial algebras that fit within the more specific framework of Ore-algebras.

1.3 Specifications

1.3.1 A flexible toolbox

As argued above our goal is to have a flexible and easy to use toolbox for computer exploration in algebraic combinatorics. This includes two clearly distinguished but closely interfaced parts: one for combinatorics and the other for algebra.

The combinatorial part should provide basic routines to deal with various combinatorial objects. As such, it is to become a large collection of relatively small functions to count, list, manipulate combinatorial objects. Most of the required combinatorial utilities are quite common (say, list all the partitions of a given integer, ...) but there are so many potential utilities that a combinatorial package will never be able to provide all of them, or at least not in an optimized form. Instead, the aim should be to make it easy for users to write such utilities as need for them arises. This includes providing versatile tools for defining and manipulating new combinatorial classes.

The algebraic part should be a sort of mecano built on top of the combinatorial part. In other packages like `ACE` or `Symmetrica`, the aim is to provide polished implementations of some specific algebras like symmetric functions. Instead, we aim at providing an unspecialized, very flexible and extensible toolbox to build new algebras, with the standard algebras being implemented as mere examples of applications of the general framework. In short, the package should try to take care of the trivial but tedious parts of the computations, letting the writer concentrate on the specific parts of his problem which are, most of the time in our experience, of combinatorial nature.

Moreover, the system should allow for a computation of the answers of natural questions such as "what is the rank of this set of vectors?" or "which elements in this combinatorial algebra are idempotents?". In such computations, one often builds large systems of equations, linear or not. Solving such systems, requires versatile general purpose computer algebra tools, like linear algebra, integration, Gröbner basis, and so on. In many occasions, the systems may become very large, requiring specialized high performance software like, for example, FGB/RS, SYNAPS, or LinBox. Interfacing with such tools should be as seamless as possible. This speaks for using a general purpose computer algebra system, that can be easily interfaced with external specialized tools.

1.3.2 A short development cycle

Another aspect of writing software for daily computer exploration is that the development cycle ought to be short. Here, genericity, flexibility, rapid prototyping, and speed of development are at a premium. Of course, efficiency is desirable but constant time factors are not necessarily so important (anyway, most of the time, the size of the studied objects grows exponentially). Optimizations are usually only really required in very specific parts (underlying linear algebra, ...); only those parts need to be optimized, after a careful analysis with a profiler. Ideally, the code should be written in such a way as to leave room for such specializations and optimizations. All of this speaks for a high-level language that allows one to write code that sticks as much as possible to the mathematical way of thinking. Of course, this does not preclude the use of external modules written in a low-level language like `C` for the critical sections, when there is a clear need for it.

1.3.3 A package designed and developed by users, for users

The package should allow different levels of use:

- Occasional usage, as a mere calculator using only predefined utilities.
- Regular usage, programming of little utilities, definition of simple new combinatorial classes and algebras;
- Intensive usage, programming of complete libraries for new combinatorial classes and algebras;
- Core hacking, implementation of generic algorithms, writing of optimized external modules (say in C), ...

For the first two levels of use, being integrated in a well-known and widely available system helps so that the user can work in his usual computing environment. This is crucial to attract new users.

As stressed above, by the very nature of the application field, any non-trivial usage involves extending the package with new utilities. To avoid duplicate work, it is essential for users to *share their code*. Ideally, the package should essentially end up acting as a *repository* of user developed routines. To this end, it needs to define a well designed framework where new utilities can be easily and quickly integrated in a natural and easy to find place. Then, defining this framework, and coordinating the developments to ensure a large scale coherency would be the main role of the core developers. Of course, the *platform* and the *development model* should encourage contributions and foster collaborations.

This aspect is particularly important for us, core developers, as our ultimate goal is to do research, not to write software. It happens that, to do this research, we need appropriate tools, and we are currently investing a lot of energy to launch this project to fulfill our own needs as well as, hopefully, other's. It is our personal hope for the near future that sharing those tools and the associated development time with others will actually save us time for more research.

1.4 Structure of this document

Apart from the preceding introduction this paper is divided into two sections. The first section is a guided tour through `MuPAD-Combinat`. After a general example of usage, we describe step by step the structure of a combinatorial class, together with two generic tools to deal with constrained list of integers and decomposable objects described by a recursive grammar (this is similar to the description of languages by context-free grammar, though here the grammar are not required to be context free). We provide in particular some examples of how to define new combinatorial classes. The next two subsection are devoted to combinatorial algebras, both predefined and new. This first part ends with a summary of the current and soon-to-be features of the package. Note that the version of this document included in the `MuPAD-Combinat` documentation provides exercises throughout this guided tour.

The second section is devoted to the design of the package. First we discuss the choice of the platform and of the development model. We explain some very basic conventions such as naming. Then we proceed with combinatorial objects and how they can be represented in a computer algebra system. We describe the design of a unified interface for various combinatorial classes on the top of which algebraic objects can be built. There, inheritance is essential to standardize and reuse code. Finally, we deal with combinatorial algebras. We describe the advantage of typing objects rather than using expressions. Then, we concentrate on the description of several implementations of free module and how the system takes care of linearity. We end up by a description of the interface for combinatorial algebras with different bases and of the mechanism for conversions between these different bases.

The suggested order of reading is to browse quickly through the guided tour (Section 2), and the design notes (Section 3, essentially the beginning of subsections *Representing combinatorial objects and classes* and *Representing combinatorial algebras*), and then to read these two sections in detail with a computer under hand to experiment with the examples.

2 A guided tour through MuPAD-Combinat

The main purpose of this package is to provide tools for manipulating combinatorial (Hopf) algebras. To setup the stage, we start this guided tour by presenting a few sample computations with two examples of such algebras. Then, we proceed by illustrating with many examples the predefined combinatorial objects and how to define new ones and the predefined combinatorial algebras and how to define new ones. We conclude this tour by a summary of the current features.

2.1 Two examples of combinatorial algebras

In this tour we assume that the up-to-date `Combinat` package has been loaded into MuPAD. Depending on your installation you may have to enter a command such as `package("Combinat"):` or `package("."):` depending on your installation.

We define a shortcut for the algebra of symmetric functions [?]:

```
>> S := examples::SymmetricFunctions():
```

We consider the three first elementary symmetric polynomials in the variables `{x1,...,x6}`:

```
>> alphabet := [ x1, x2, x3, x4, x5, x6]:
   e1 := expand(S::e([1])(alphabet));

                     x1 + x2 + x3 + x4 + x5 + x6

>> e2 := expand(S::e([2])(alphabet));

 x1 x2 + x1 x3 + x1 x4 + x2 x3 + x1 x5 + x2 x4 + x1 x6 +

    x2 x5 + x3 x4 + x2 x6 + x3 x5 + x3 x6 + x4 x5 + x4 x6 +

    x5 x6

>> e3 := expand(S::e([3])(alphabet))

 x1 x2 x3 + x1 x2 x4 + x1 x2 x5 + x1 x3 x4 + x1 x2 x6 +

    x1 x3 x5 + x2 x3 x4 + x1 x3 x6 + x1 x4 x5 + x2 x3 x5 +

    x1 x4 x6 + x2 x3 x6 + x2 x4 x5 + x1 x5 x6 + x2 x4 x6 +

    x3 x4 x5 + x2 x5 x6 + x3 x4 x6 + x3 x5 x6 + x4 x5 x6
```

As one can see, the system distinguishes between abstract (or “symbolic”) symmetric functions such as `S::e([3])`, and the expansion of them as symmetric polynomials on the alphabet, stored here in variables such as `e3`. The call `expand(f(alphabet))` for an abstract symmetric function `f` actually expands the corresponding symmetric polynomial over the alphabet.

Computing the product of two such symmetric polynomials yields a huge polynomial which is not quite practical to manipulate:

```
>>   expand(e2*e3)

 10 x1 x2 x3 x4 x5 + 10 x1 x2 x3 x4 x6 + 10 x1 x2 x3 x5 x6 +

    10 x1 x2 x4 x5 x6 + 10 x1 x3 x4 x5 x6 + 10 x2 x3 x4 x5 x6 +

                    2
    3 x1 x2 x3 x4  + ... (one page of output)

            2   2
    x1 x2  x3  + ... (another page of output)
```

Instead, if we use the symmetries, the previous product can be expressed as compactly as:

```
>>   S::m( S::e([2]) * S::e([3]) );

       10 m[1, 1, 1, 1, 1] + 3 m[2, 1, 1, 1] + m[2, 2, 1]
```

Here, `m[2, 1, 1, 1]` denotes the monomial symmetric function $m_{2,1,1,1}$ obtained by summing all the monomials with one variable elevated to the power 2 and three variables to the power 1. The product has been calculated at the level of abstract symmetric functions without expanding the polynomials, which is much faster and requires substantially less memory.

Hence, symmetric functions provide a typical example of *combinatorial algebra* whose bases are indexed by combinatorial objects (partitions), and where we want to compute efficiently.

As another typical example, we are currently working [?, ?] on the so-called Loday-Ronco algebra [?], which is in particular of interest for theoretical physicists [?, ?]. It is implemented as a combinatorial algebra having binary trees as basis. Here we use computation in the fundamental basis denoted by `p`.

```
>> LRA := examples::LodayRoncoAlgebra():
```

For example, take the two following trees:

```
>> t1 := LRA::p(combinat::binaryTrees::unrank(6, 4))

                            p/  o   \
                             | / \  |
                             \    \ /
```

```
>> t2 := LRA::p(combinat::binaryTrees::unrank(26, 5))

                            p/   o  \
                             |  / \ |
                             \ /\   /
```

Technically, `combinat::binaryTrees::unrank(k,n)` returns the k-th tree with n nodes, while `LRA::p(t)` returns the basis element of LRA::p indexed by t.

You can make a formal linear combination of `t1` and `t2`:

```
>> 2*t2 + 3/4*t1

                   2 p/   o  \ + 3/4 p/  o   \
                      |  / \ |        | / \  |
                      \ /\   /        \    \ /
```

or take their product:

```
>> t2*t1

 p/   o     \ + p/   o    \ + p/    o   \ + p/   o    \ +
  |  / \    |    |  / \   |    |   / \  |    |  / \   |
  | /\  \   |    | /\ /\  |    |  /\  \ |    | /\ /\  |
  |     /\  |    \    \ \ /    \ /\ \   /    \   /  \ /
  \       \ /

    p/    o   \ + p/     o   \
     |  / \   |    |    / \  |
     |  /\  \ |    |   /   \ |
     \ /\/    /    |  /\     |
                   \ /\      /
```

Here is a more complicated product in this algebra:

```
>> (2*t2 + 3/4*t1) * t2

 3/4 p/  o     \ + 3/4 p/  o    \ + 3/4 p/  o   \ +
      | / \    |        | / \   |        | / \  |
      |    \   |        |    \  |        |  / \ |
      |     \  |        |    /\ |        |  \   |
      |     /\ |        |    \  |        |   \  |
      \    /\  /        \    /\ /        \   /\ /
```

```
3/4 p/   o   \ + 3/4 p/  o    \ + 3/4 p/  o   \ +
     |  /  \ |        | / \   |        | / \  |
     | /\    |        |    \  |        |   /\ |
     |   \   |        |    /\ |        |   \  |
     |    \  |        |   /\  |        |   /\ |
     \    /\ /        \   \   /        \   \  /

3/4 p/   o  \ + 3/4 p/  o   \ + 3/4 p/   o  \ +
     |  / \ |        | / \  |        |  / \ |
     | /\   |        |   /\ |        | /\   |
     |   \  |        |  /\  |        |  /\  |
     |   /\ |        |  \   |        |  \   |
     \   \  /        \   \  /        \   \  /

3/4 p/    o  \ + 3/4 p/  o    \ + 3/4 p/  o   \ +
     |   / \ |        | / \   |        | / \  |
     |  / \  |        |    \  |        |   /\ |
     | /\    |        |    /\ |        |   \  |
     |   \   |        |   /\  |        |   /\ |
     \    \  /        \  /    /        \  /   /

3/4 p/   o  \ + 3/4 p/  o   \ + 3/4 p/   o  \ +
     |  / \ |        | / \  |        |  / \ |
     | /\   |        |   /\ |        | /\   |
     |   \  |        |  /\  |        |  /\  |
     |   /\ |        |  \   |        |  \   |
     \  /   /        \  /   /        \  /   /

3/4 p/    o  \ + 3/4 p/  o   \ + 3/4 p/   o  \ +
     |   / \ |        | / \  |        |  / \ |
     |  /\   |        |   /\ |        | /\   |
     | /\    |        |  /\  |        |  /\  |
     |   \   |        | /    |        | /    |
     \   /   /        \ \    /        \ \    /

3/4 p/    o  \ + 3/4 p/     o  \ + 2 p/   o    \ +
     |   / \ |        |    / \ |      |  / \   |
     |  /\   |        |   /\   |      | /\  \  |
     | /\    |        |  /     |      |     /\ |
     |  /    |        | /\     |      \    /\  /
     \  \    /        \   \    /

2 p/   o   \ + 2 p/    o  \ + 2 p/   o   \ + 2 p/    o  \ +
   |  / \  |      |   / \ |      |  / \  |      |   / \ |
   | /\ /\ |      |  /\   |      | /\ /\ |      |  /\   |
   |    \  |      | /\ \   |      |    /\  |      | /\/\   |
   \    /\ /      \    /\ /      \   \   /      \   \   /
```

```
2 p/     o  \ + 2 p/   o   \ + 2 p/    o  \ +
   |    / \ |      |  / \  |      |   / \ |
   |   /\   |      | /\ /\ |      |  / \  |
   |  /\    |      |   /\  |      | /\ /\ |
   \ /\ \   /      \  /    /      \   /   /

2 p/     o  \ + 2 p/      o  \
   |    / \ |      |     / \ |
   |   /\   |      |    /\   |
   |  /\    |      |   /     |
   \ /\/    /      |  /\     |
                   \ /\      /
```

2.2 MuPAD-Combinat, step by step

We now describe in more detail how all of this works. In the following, we assume that the package MuPAD-Combinat has been loaded into MuPAD. We also assume that the reader is somewhat familiar with the MuPAD syntax We refer to the MuPAD tutorial for details. Technicalities can be safely ignored in a first reading; they will be better understood after the explanations in the design notes.

2.2.1 Using predefined combinatorial functions and classes

For shortening the notations, we export the library combinat:

```
>>   export(combinat):
```

We start by some sample applications at random. We compute the first terms of the famous Catalan sequence, we generate the Cartesian product of three lists, we compute all permutations of the numbers $1, 2, 3$, and we ask for all sub-words of the word [a, b, c, d]:

```
>> catalan(i) $ i = 0..10

         1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796

>> cartesianProduct::list([1,2,3],[a,b],[i,ii,iii])

 [[1, a, i], [1, a, ii], [1, a, iii], [1, b, i], [1, b, ii],

    [1, b, iii], [2, a, i], [2, a, ii], [2, a, iii], [2, b, i],

    [2, b, ii], [2, b, iii], [3, a, i], [3, a, ii],

    [3, a, iii], [3, b, i], [3, b, ii], [3, b, iii]]
```

```
>> permutations::list([1, 2, 3])

 [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2],

    [3, 2, 1]]

>> subwords::list([a,b,c,d])

 [[], [a], [b], [c], [d], [a, b], [a, c], [a, d], [b, c],

    [b, d], [c, d], [a, b, c], [a, b, d], [a, c, d], [b, c, d],

    [a, b, c, d]]
```

We turn now to various combinatorial classes. In short, a *combinatorial class* is a set of related combinatorial objects, like the set of all integer partitions. For every such classes, there is a sub-library of `combinat`. We can use the library `combinat::partitions` to list all the integer partitions of 5:

```
>> partitions::list(5)

 [[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1],

    [1, 1, 1, 1, 1]]
```

Let us draw the partition `[3,2]` using boxes (French or Cartesian notation):

```
>> partitions::printPretty([3, 2])

                              +---+---+
                              |   |   |
                              +---+---+---+
                              |   |   |   |
                              +---+---+---+
```

We can fill those boxes with the numbers `1,2,3,4,5` so that the numbers are increasing along rows and columns to obtain so-called *standard tableaux*. Here are all the standard tableaux of shape `[3,2]`:

```
>> map(tableaux::list([3, 2]), tableaux::printPretty)

 -- +---+---+       +---+---+       +---+---+       +---+---+
 |  | 4 | 5 |       | 3 | 5 |       | 2 | 5 |       | 3 | 4 |
 |  +---+---+---+   +---+---+---+   +---+---+---+   +---+---+---+
 |  | 1 | 2 | 3 |,  | 1 | 2 | 4 |,  | 1 | 3 | 4 |,  | 1 | 2 | 5 |,
 -- +---+---+---+   +---+---+---+   +---+---+---+   +---+---+---+
    +---+---+     --
    | 2 | 4 |      |
    +---+---+---+  |
    | 1 | 3 | 5 |  |
    +---+---+---+ --
```

Ordered trees are another typical combinatorial class. Here are all the trees on four nodes:

```
>> trees::list(4)

                   --  o ,  o ,  o ,  o , o --
                   |  /|\  / \  / \   |   |  |
                   |        |  |    / \  |  |
                   --                    | --
```

and here are some more trees:

```
>> trees::list(6)

 --   o  ,   o  ,   o  ,  o  ,  o ,   o  ,  o ,   o  ,  o ,
 |  //|\\  // \\  // \\  /|\   /|\  // \\  /|\  / | \  /|\
 |             |     |     /\    |   |      ||   / \    |
 |                               |                      |
 |
 --

     o  ,  o  ,  o  ,  o  ,  o ,   o  ,  o ,  o ,  o  ,  o ,
    / \   / \   / \   / \   / \  // \\  /|\  /|\  / \   / \
     /|\    /\    /\    |     |  |      | |  ||   | /\  | |
             |    |    / \    |                           |
                              |

      o ,   o ,  o ,  o ,   o ,   o ,   o ,   o ,  o ,   o  ,
     /|\   / \  /|\  / \   / \   / \   / \   / \  / \    |
    /\    /\ |  |    | |  /|\   /\    /\     |    |    // \\
                |    |           |    |     / \   |
                                                  |

     o ,  o ,  o  ,  o ,  o ,  o ,   o ,  o ,  o ,  o ,  o ,
     |    |    |     |    |    |     |    |    |    |    |
    /|\  /|\  / \   / \  /|\  / \   / \  / \   |    |    |
      |   |     /\    |  |    | |  /\    |    /|\  / \  / \
                      |                  |               |  |
     o , o --
     |   |  |
     |   |  |
     |   |  |
    / \  |  |
         | --
```

All the sub-libraries of `combinat` share a standardized interface. Let us look in more detail at the library `combinat::partitions`. We can count partitions:

```
>> partitions::count(i) $ i = 0..10

                1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42
```

list them under some extra conditions (here we list the partitions of 5 whose length is between 2 and 3):

```
>> partitions::list(5, MinLength = 2, MaxLength = 3);

               [[4, 1], [3, 2], [3, 1, 1], [2, 2, 1]]
```

or compare them (lexicographically):

```
>> bool(partitions::_less([3, 1], [2, 2]))

                                FALSE
```

An important feature of `MuPAD-Combinat` are the so-called *generators*, which allow programs to run through huge lists of combinatorial objects without expanding the full lists into memory. Technically, a generator is a function `g` such that each call `g()` returns either a new object, or FAIL if no more objects are available. Let us build a generator for the partitions of 4:

```
>> g := partitions::generator(4):
```

Here is the first partition of 4:

```
>> g()

                                 [4]
```

Here is the second partition of 4:

```
>> g()

                               [3, 1]
```

And here are the remaining ones:

```
>> g(), g(), g(), g()
```

```
                [2, 2], [2, 1, 1], [1, 1, 1, 1], FAIL
```

Generators come in handy when you want to work with the 53174 partitions of 42:

```
>> g := partitions::generator(42):
   g(), g(), g(), g(), g(), g()

      [42], [41, 1], [40, 2], [40, 1, 1], [39, 3], [39, 2, 1]
```

Most of the sub-libraries of `combinat` provide such generators.

Whenever possible (i.e. when it does not harm the computational complexity), we focus on providing the user with generic tools that cover many kinds of applications. For example, the libraries for partitions, integer vectors, and compositions share a very similar interface:

```
>> integerVectors::list(10, 3, MinPart = 2, MaxPart = 5,
                                 Inner = [2, 4, 2])
```

(Note: `Inner = [2, 4, 2]` means that the three parts should be respectively at least 2, 4, and 2).

```
     [[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]

>> compositions::list(5, MaxPart = 3, MinPart = 2,
                         MinLength = 2, MaxLength = 3)

                         [[3, 2], [2, 3]]

>> partitions::list(5, MaxSlope = -1)

                       [[5], [4, 1], [3, 2]]
```

Those libraries actually use internally the same computational engine `combinat::integerListsLexTools`:

```
>> partitions::list(9, MinPart = 2, MaxPart = 5)

     [[5, 4], [5, 2, 2], [4, 3, 2], [3, 3, 3], [3, 2, 2, 2]]

>> integerListsLexTools::list(9, 0, infinity, 2, 5, -infinity, 0)

     [[5, 4], [5, 2, 2], [4, 3, 2], [3, 3, 3], [3, 2, 2, 2]]
```

In fact, the algorithm of `combinat::integerListsLexTools` could also be used to generate Motzkin and Dyck words, etc.

In the same spirit, instead of implementing a specific generator for standard tableaux, we implemented a generator for the linear extensions of a poset. We already reused this generator internally for generating standard binary search trees, and it could be reused as well for generating standard skew tableaux, standard ribbons, and so on.

We also incorporated and extended the former `CS` library by S. Corteel, A. Denise, I. Dutour, and P. Zimmermann. This library allows one to manipulate combinatorial classes that can be defined by a deterministic grammar. Here we consider words of A's and B's without two consecutive B's. Such a word is

- either void;
- either the word ”B”;
- either a word ending by a ”A”;
- or finally a word ending by ”AB”.

In the two later case, the beginning of the word is of the same type. This can be used to build a recursion process for generating recursively all such words. Such a recursion process is called a grammar. Note that this process leads to an unambiguous grammar, that is each word is appears one and only one in the generation process, otherwise said the previous four cases are mutually exclusive. This is translated in MuPAD by

```
>> fiWords := decomposableObjects(
       [FiWords = Union(Epsilon,
                        Atom(B),
                        Prod(FiWords, Atom(A)),
                        Prod(FiWords, Atom(A), Atom(B)))
       ]):
```

(Note: an `Epsilon` is an object of size 0 while an `Atom` is an object of size 1).

```
>> fiWords::list(4)

 [Prod(Prod(Prod(B, A), A), A), Prod(

    Prod(Prod(Prod(Epsilon, A), A), A), A),

    Prod(Prod(Prod(Epsilon, A, B), A), A),

    Prod(Prod(B, A, B), A), Prod(Prod(Prod(Epsilon, A), A, B), A

    ), Prod(Prod(B, A), A, B), Prod(Prod(Prod(Epsilon, A), A),

    A, B), Prod(Prod(Epsilon, A, B), A, B)]
```

The result is not very readable, but this can be fixed by a quick substitution:

```
>> map(fiWords::list(4), p -> [eval(subs(p, Prod = id,
                                        Epsilon = null() ))])
 [[B, A, A, A], [A, A, A, A], [A, B, A, A], [B, A, B, A],

    [A, A, B, A], [B, A, A, B], [A, A, A, B], [A, B, A, B]]
```

Alternatively, we could have provided some extra rewriting rules within the grammar. Notice that the preceding grammar generates the words in an order that is not so obvious. With some small reordering of the grammar, it is possible to ensure that the words are generated in the lexicographic order:

```
>> fiWords := combinat::decomposableObjects(
       [FiWords    = Alias(FiWordsRec, DOM_LIST),
        FiWordsRec = Union(Epsilon(),
            Alias(Prod(Atom(A), FiWordsRec), op),
            Atom(B),
            Alias(Prod(Atom(B), Atom(A), FiWordsRec), op)
        )
       ]):
   fiWords::list(4);
 [[A, A, A, A], [A, A, A, B], [A, A, B, A], [A, B, A, A],

    [A, B, A, B], [B, A, A, A], [B, A, A, B], [B, A, B, A]]
```

This seems to work nicely. Let us count those words:

```
>> fiWords::count(i) $ i = 0..10
              1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
```

You will certainly recognize the Fibonacci sequence. Not quite a surprise, the recurrence relation can be seen right away from the grammar. Actually, this recurrence relation is automatically determined by the library, and used for counting efficiently:

```
>> fiWords::recurrenceRelation() = 0
                  u(n - 1) - u(n) + u(n - 2) = 0
```

This also applies for several libraries which are based on `combinat::decomposableObjects`. For example, here is the recurrence relation for binary trees:

```
>> collect(binaryTrees::grammar::recurrenceRelation(),
           [u(n), u(n-1)], factor) = 0
              u(n) (n + 1) - 2 u(n - 1) (2 n - 1) = 0
```

2.2.2 Defining new combinatorial classes

For shortening the notations, we export the library `combinat`:

```
>>   export(combinat):
```

Let us define one of the most trivial combinatorial classes:

```
>> domain oddIntegers
       // This is a graded combinatorial class:
       category Cat::GradedCombinatorialClass;
       // This is a domain (not a library):
       inherits Dom::BaseDomain;
       // This is a facade domain:
       axiom Ax::systemRep;

       info := "The class of non negative odd integers";

       isA := n -> bool(testtype(n, Type::PosInt) and
                        n mod 2 <> 0);
       // The size of an odd integer is itself
       size  := n -> n;
       count := n -> if n mod 2 = 1 then 1 else 0 end_if;
       list  := n -> if n mod 2 = 1 then [n] else [] end_if;

       // No need to define generator;
       // it is defined via list by default
   end_domain:
```

In a first approximation, the three lines `inherits`, `category`, and `axiom` may be safely ignored and kept verbatim. For a deeper understanding, we strongly recommend to read the detailed explanations about the implementation of combinatorial classes in the design notes.

Now, this combinatorial class can be used like all the others:

```
>> testtype(x, oddIntegers), testtype(-3, oddIntegers),
   testtype(2, oddIntegers), testtype(3, oddIntegers);

                     FALSE, FALSE, FALSE, TRUE

>> oddIntegers::count(3);

                                 1

>> oddIntegers::list(5)

                                [5]
```

In particular, it can be used as building block for constructing new combinatorial classes like, say, integer partitions with odd parts:

```
>> oddPartsPartitions := combinat::decomposableObjects
                              ([P = Multiset(oddIntegers)]):

>> oddPartsPartitions::list(5)

    [Multiset(5), Multiset(1, 1, 3), Multiset(1, 1, 1, 1, 1)]
```

It is often practical to define a sub-class of an existing class. Here we show how to define the class of the permutations of [1,2,3]:

```
>> domain permutationsOf123
       category Cat::CombinatorialClass;
       // Inherits all the methods from combinat::permutations
       inherits permutations;
       axiom    Ax::systemRep;

       info := "the class of the permutations of [1,2,3]";

       // Redefinition of isA, count and generator
       isA := (p) -> permutations::isA(p, [1,2,3]);
       count     := () -> permutations::count([1,2,3]);
       generator := () -> permutations::generator([1,2,3]);

       // No need to redefine list, since it is
       // defined via generator by default
   end_domain:
```

Let us use this new combinatorial class:

```
>> testtype(x,            permutationsOf123),
   testtype([1, 2, 3, 4], permutationsOf123),
   testtype([1, 2, 2],    permutationsOf123),
   testtype([1, 3, 2],    permutationsOf123);

                    FALSE, FALSE, FALSE, TRUE

>> permutationsOf123::count();

                                6

>> permutationsOf123::list()

 [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2],

    [3, 2, 1]]
```

Note: instead of implementing `permutationsOf123` by hand, we could have alternatively used the generic utility `combinat::subClass`; it allows one to automatically define a sub-class of an existing combinatorial class by providing extra parameters to be passed down to all the methods `count`, `list`, etc.:

```
>> permutationsOf123 := subClass(permutations, Parameters = 3):
   permutationsOf123::list()

 [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2],

    [3, 2, 1]]
```

To conclude, we define the combinatorial class of Fibonacci words. Essentially, we reuse the definition of `fiWords` above, and wrap it into a domain to add type checking:

```
>> domain FibonacciWords
       // The objects of this class are defined by a grammar
       category Cat::DecomposableClass;
       inherits Dom::BaseDomain;
       axiom    Ax::systemRep;

       info := "the class of Fibonacci words";

       // The domain of the elements of this class
       domtype := DOM_LIST;

       // The type of the elements of this class:
       // a procedure that tests if w is a Fibonacci word
       isA := proc(w : Type::AnyType,
                   size = 0 : Type::NonNegInt)
           local i;
           begin
               if domtype(w)<>DOM_LIST then return(FALSE); end_if;
               for i from 1 to nops(w) do
                   if (w[i]<>A and w[i]<>B) or
                      (i<nops(w) and w[i]=B and w[i+1]=B) then
                       return(FALSE);
                   end_if;
               end_for;
               if args(0) = 1 then TRUE else
                   bool(nops(w) = size);
               end_if;
           end_proc;
```

```
        // The size of a Fibonacci word is its length
        size := nops;

        // The grammar which defines the objects of this class
        grammar := decomposableObjects(
            [FiWords    = Alias(FiWordsRec, DOM_LIST),
             FiWordsRec = Union(
                 Epsilon(),
                 Alias(Prod(Atom(A), FiWordsRec), op),
                 Atom(B),
                 Alias(Prod(Atom(B), Atom(A), FiWordsRec), op))]);
    end_domain:
```

Now, we can do type checking with this domain:

```
>> testtype(x, FibonacciWords),
   testtype([A, B, C], FibonacciWords),
   testtype([A, B, B], FibonacciWords),
   testtype([A, B, A], FibonacciWords)

                      FALSE, FALSE, FALSE, TRUE
```

And of course, we can still use all the previous functionalities of `fiWords`:

```
>> FibonacciWords::list(4);

 [[A, A, A, A], [A, A, A, B], [A, A, B, A], [A, B, A, A],

    [A, B, A, B], [B, A, A, A], [B, A, A, B], [B, A, B, A]]

>> FibonacciWords::count(4);

                                   8
```

And finally if we wanted to extract the second element of the list without expanding it we can ask for the words of rank 2 in the list:

```
>> FibonacciWords::unrank(2, 4)

                             [A, A, A, B]
```

2.2.3 Using predefined combinatorial algebras

We now demonstrate how to do sample computations with predefined combinatorial algebras, starting with the algebra of symmetric functions. Note that we really consider those predefined algebras as mere examples of use of this package; important and useful examples of course, but just examples.

We define the ring of symmetric functions over the rational numbers:

```
>> S := examples::SymmetricFunctions(Dom::Rational);

            examples::SymmetricFunctions(Dom::Rational)
```

This ring has several remarkable families like the symmetric *power-sums* p_k: recall that the symmetric power-sum p_k expands on any given specified *alphabet* (i.e. set of variables) as the sum of all the variables elevated to the power k; furthermore, given a partition $\lambda := (\lambda_1, \ldots, \lambda_k)$, the product of $p_{\lambda_1} \ldots p_{\lambda_k}$ is denoted by p_λ:

```
>> p1 := S::p([1]);
   p1([x, y, z]);

                                p[1]

                             x + y + z

>> p2 := S::p([2]);
   p2([x, y, z])

                                p[2]

                             2    2    2
                            x  + y  + z

>> p421 := S::p([4, 2, 1]);
   p421([x,y,z])

                             p[4, 2, 1]

                 2    2    2     4    4    4
               (x  + y  + z ) (x  + y  + z ) (x + y + z)
```

Note: the product being commutative, the order in which the terms appear in the expansion above depends on MuPAD internal order, and is mathematically irrelevant.

Actually, the ring of symmetric functions is the free commutative algebra on the symmetric power-sums:

```
>> p2 * p1 * p2 * p421

                          p[4, 2, 2, 2, 1, 1]
```

Note that any expression is immediately expanded by the system:

```
>> (p421 + 3*p2)*(1/4*p1 - p2)

   1/4 p[4, 2, 1, 1] - 3 p[2, 2] - p[4, 2, 2, 1] + 3/4 p[2, 1]
```

This happens because the call `S::p([1])` returns a typed object, for which the standard arithmetic operators are overloaded:

```
>> domtype(p1);

   examples::SymmetricFunctionsTools::powersum(Dom::Rational)

>> S::p

   examples::SymmetricFunctionsTools::powersum(Dom::Rational)
```

That is, `S::powersum` (or `S::p` for short) really represents the domain of symmetric functions expanded on the power-sums basis. If at some time you do not want the expansion to take place, the objects can always be converted to expressions:

```
>> f := (expr(p421) + 3*expr(p2))*(1/4*expr(p1) - expr(p2))

                                / p[1]        \
            (3 p[2] + p[4, 2, 1]) | ---- - p[2] |
                                \   4         /
```

This expression can by converting back to a symmetric function :

```
>> S(f);

   1/4 p[4, 2, 1, 1] - 3 p[2, 2] - p[4, 2, 2, 1] + 3/4 p[2, 1]
```

To do this we use the following small trick because of the indexed notation for the basis elements.

```
>> eval(subs(f, p = S::p::domainWrapper))

   1/4 p[4, 2, 1, 1] - 3 p[2, 2] - p[4, 2, 2, 1] + 3/4 p[2, 1]
```

The explanation of the trick is that `S::p::domainWrapper` is a special MuPAD object which, when used as `S::p::domainWrapper[3,2]`, returns a call to `S::p([3,2])`.

Of course, `examples::SymmetricFunctions` provides the other classical bases of symmetric functions, like the elementary symmetric functions `S::e`, the monomial symmetric functions `S::m`, the homogeneous symmetric functions `S::h`, the Schur functions `S::s`, etc.:

```
>> expand(S::e([2])([x,y,z]))

                          x y + x z + y z

>> expand(S::m([2, 1])([x,y,z]))

                2    2         2    2         2    2
             x y  + x  y + x z  + x  z + y z  + y  z
```

```
>> expand(S::h([2])([x,y,z]))

                                       2    2    2
                 x y + x z + y z + x  + y  + z

>> expand(S::s([2])([x,y,z]))

                                       2    2    2
                 x y + x z + y z + x  + y  + z
```

Here is how to convert from one basis to the other:

```
>> f := S::p([4]);
   S::e(f);
   S::h(f);
   S::s(f);
   S::m(f)

                              p[4]

  e[1, 1, 1, 1] - 4 e[2, 1, 1] + 4 e[3, 1] - 4 e[4] + 2 e[2, 2]

 - h[1, 1, 1, 1] + 4 h[2, 1, 1] - 4 h[3, 1] + 4 h[4] - 2 h[2, 2]

       - s[1, 1, 1, 1] + s[2, 1, 1] - s[3, 1] + s[4]

                              m[4]
```

When multiplying two symmetric functions which are not expressed in the same basis, the system will make an implicit conversion, and return the result in one or the other of the two bases:

```
>> S::m([2]) * S::s([2])

            m[2, 1, 1] + m[3, 1] + m[4] + 2 m[2, 2]
```

If you want to force the product to be done on a given basis, you can call the proper conversion explicitly:

```
>> S::s(S::m([2])) * S::s([2]);

                - s[2, 1, 1] + s[4] + s[2, 2]
```

Now, we can combine everything, and do some complicated calculation:

```
>> S::p( S::m([1]) * ( S::e([3])*S::s([2]) + S::s([3]) ) )
```

```
 1/12 p[1, 1, 1, 1, 1, 1] + 1/6 p[3, 2, 1] -

    1/6 p[2, 1, 1, 1, 1] + 1/6 p[3, 1, 1, 1] -

    1/4 p[2, 2, 1, 1] + 1/6 p[1, 1, 1, 1] + 1/2 p[2, 1, 1] +

    1/3 p[3, 1]
```

Finally, there is some basic support for the Hall-Littlewood functions, in the P and Q' basis, which we demonstrate now. We need to take some ground field which contains the parameter t of those functions. The simplest (and actually most efficient with the current MuPAD version), is to take the full field of expressions as coefficient ring:

```
>> S := examples::SymmetricFunctions(Dom::ExpressionField()):
```

Here is the Hall-Littlewood function $Q'_{(3,2,1,1)}$:

```
>> el := S::QP([3, 2, 1, 1])

                                  QP[3, 2, 1, 1]
```

The expansion of `el` in terms of Schur functions reads as:

```
>> S::s(el)

                              2
 t s[3, 2, 2] + (t + t ) s[3, 3, 1] + t s[4, 1, 1, 1] +

      2     3     4               2     3     4
    (t  + t  + t ) s[4, 3] + (t  + t  + t ) s[5, 1, 1] +

        3     4     5                 4     5     6
    (2 t  + t  + t ) s[5, 2] + (t  + t  + t ) s[6, 1] +

     7                                    2     3
    t  s[7] + s[3, 2, 1, 1] + (t + 2 t  + t ) s[4, 2, 1]
```

The expansion of `el` on the alphabet (q, qt) reads as:

```
>> expand(el([q, q*t]))

    7  5        7  6         7  7        7  8        7  9        7  10
 4 q  t  + 7 q  t  + 10 q  t  + 9 q  t  + 6 q  t  + 5 q  t   +

       7  11        7  12     7  13     7  14
    3 q  t   + 2 q  t   + q  t   + q  t
```

2.2.4 Defining new combinatorial algebras

We now turn to the central feature of the `MuPAD-Combinat` package: the ability to easily implement new combinatorial algebras. We start by the free associative algebra over the rational numbers generated by non commutative letters $a, b, c, d, \ldots$. Its basis is indexed by words, and the product of two basis elements is obtained by concatenating the corresponding words:

```
>> domain FreeAlgebra
       inherits Dom::FreeModule(Dom::Rational, combinat::words);
       category Cat::AlgebraWithBasis(Dom::Rational);

       one           := dom::term([]);
       mult2Basis    := dom::term @ _concat;
   end_domain:
```

We will explain the bits of this definition in a minute after a few examples of use. Let us define two elements of the free algebra:

```
>> x := FreeAlgebra([a, b, c]);
   y := FreeAlgebra([d, e])

                              B([a, b, c])

                                B([d, e])
```

The `B` just stands for the name of the basis. We can compute linear combinations and products of `x` and `y`:

```
>> 3 * x;
   x + y;
   x * y

                             3 B([a, b, c])

                       B([d, e]) + B([a, b, c])

                           B([a, b, c, d, e])
```

Here is a more complicated expression:

```
>> x * (2*x + y) + (3 + y/2)^2

 1/4 B([d, e, d, e]) + 2 B([a, b, c, a, b, c]) +

    B([a, b, c, d, e]) + 3 B([d, e]) + 9 B([])
```

Note how the `3` in the expression is automatically converted into an element of the domain; declaring that `FreeAlgebra` was an algebra (with a unit) automatically defined the natural embedding of the coefficient ring into it.

We turn to the explanation of the implementation of `FreeAlgebra` above. The line

```
   domain FreeAlgebra
```

states that we are defining a new *domain* called `FreeAlgebra` (a new class in the usual object oriented terminology).

```
    inherits Dom::FreeModule(Dom::Rational, combinat::words);
```

lets `FreeAlgebra` inherit its implementation from the free module over the rationals (`Dom::Rational`) with basis indexed by words (`combinat::words`).

```
    category Cat::AlgebraWithBasis(Dom::Rational);
```

states that `FreeAlgebra` is actually an algebra with a distinguished basis; this allows one, in particular, to define the multiplication by linearity on the basis.

```
    one           := dom::term([]);
```

defines that the unit of `FreeAlgebra` is the empty word (`dom` refers to the domain being defined, and `dom::term` is a constructor that takes an element of the basis, and returns it as an element of the domain). Finally,

```
    mult2Basis    := dom::term @ _concat;
```

states that two elements of the basis are multiplied by concatenating them, and making an element of the domain with the result (`@` denotes the composition of functions). That's it.

Let us define the free commutative algebra on the letters $a, b, c, \ldots$:

```
>> domain FreeCommutativeAlgebra
       inherits  Dom::FreeModule(Dom::Rational, combinat::words);
       category Cat::AlgebraWithBasis(Dom::Rational);

       one     := dom::term([]);
       straightenBasis := dom::term @ sort;
       mult2Basis      := dom::straightenBasis @ _concat;
   end_domain:
```

Note that we cheated a little bit: we declared that the basis of `FreeCommutativeAlgebra` consisted of words, whereas it really consists of words up to permutation of its letters: `B([a,b])` and `B([b,a])` represent the same element of the algebra. A careful implementation should define the combinatorial class of words up to permutation, and use it as the basis of FreeCommutativeAlgebra.

To enforce the uniqueness of the representation, we straighten the words in the basis by sorting them. This is the job of the straightenBasis constructor.

```
>> x := FreeCommutativeAlgebra([a, b]);
   y := FreeCommutativeAlgebra([c, b, a])

                                 B([a, b])

                               B([a, b, c])
```

The product of two words is then defined by concatenating them and straightening the result:

```
>> x * y;
   y * x

                            B([a, a, b, b, c])

                            B([a, a, b, b, c])
```

If efficiency was at a premium, instead of sorting the concatenation of the two lists, we could have used the MuPAD function listlib::merge which merges sorted lists.

Note that two elements of FreeCommutativeAlgebra and of FreeAlgebra may happen to be printed out the same way:

```
>> x := FreeAlgebra([a]);
   y := FreeCommutativeAlgebra([a])

                                  B([a])

                                  B([a])
```

and even share the exact same internal representation:

```
>> bool(extop(x) = extop(y))

                                   TRUE
```

However, they are not equal, because they are not in the same domain:

```
>> bool(x = y);
   domtype(x), domtype(y)

                                   FALSE

                  FreeAlgebra, FreeCommutativeAlgebra
```

So, even if they share the same name of basis, there is no risk of confusion; for example we are not allowed to multiply them together:

```
>> x * y

 Error: Don't know how to multiply a FreeAlgebra by a FreeCommu\
 tativeAlgebra
```

Of course, this is still confusing for the user. He or she may always customize the basis names (as many other things) at any time should he or she wish to do so:

```
>> FreeAlgebra::basisName            := hold(T):
   FreeCommutativeAlgebra::basisName := hold(S):
   x, y

                         T([a]), S([a])
```

Here, `T` stands for "Tensor algebra", while `S` stands for "Symmetric algebra". The `hold` is there for safety, in case one of the identifiers `T` or `S` is assigned a value.

We can define the natural evaluation morphism from `FreeAlgebra` to `FreeCommutativeAlgebra` by linearity on the words; a word itself is simply sorted, and converted into an element of `FreeCommutativeAlgebra`:

```
>> evaluation := operators::makeLinear
                    (FreeCommutativeAlgebra::term @ sort,
                     Source   = FreeAlgebra,
                     ImageSet = FreeCommutativeAlgebra):
```

Let us apply this morphism to the sum of two words which only differ by a permutation:

```
>> x := FreeAlgebra([c, b, a]) + FreeAlgebra([c, a, b]);

                    T([c, a, b]) + T([c, b, a])

>> evaluation(x);

                         2 S([a, b, c])
```

The evaluation morphism is actually quite canonical, so it can make sense to declare it as a conversion to the system. This can be achieved with the `operators::overloaded::declareConversion` function:

```
>> operators::overloaded::declareConversion(
       FreeAlgebra, FreeCommutativeAlgebra, evaluation):
   FreeCommutativeAlgebra(x)

                         2 S([a, b, c])
```

Here, the conversion has been declared as implicit. If an expression mixes elements of `FreeAlgebra` and `FreeCommutativeAlgebra`, the former are automatically converted into `FreeCommutativeAlgebra`:

```
>> FreeCommutativeAlgebra([a, b]) + FreeAlgebra([c,b,a])

                    S([a, b]) + S([a, b, c])
```

Of course, such a feature is questionable. Depending on the context, it can prove very practical, or on the contrary dangerous. The user is the only judge, and she or he can restrict the scope of this conversion by using the `Explicit` option. In this case, the conversion will only be applied if requested explicitly by `convert` or by `new`:

```
>> operators::overloaded::declareConversion(
        FreeAlgebra, FreeCommutativeAlgebra,
        evaluation, Explicit):
   FreeCommutativeAlgebra(x);

                          2 S([a, b, c])

>> FreeCommutativeAlgebra([a, b]) + FreeAlgebra([c,b,a])

 Error: Don't know how to add a FreeCommutativeAlgebra and a Fr\
 eeAlgebra
```

Typically, for symmetric functions, we only provided explicit conversions to construct symmetric functions from partitions because those conversions are not canonical at all: the Schur function `s[3,2,1]` obtained by converting the partition `[3,2,1]` has nothing to do with the elementary function `e[3,2,1]`. We refer to the design notes and to the documentation of the `operators::overloaded` library for details on the mechanism we use for defining automatic conversions and overloaded operators and functions. Note that it is not (yet) completely possible to declare new conversions as above when the target domain of the conversion is one of the predefined domains of the MuPAD library.

2.2.5 Combinatorial algebras with several representations

As usual, for shortening the notations, we export the library `combinat`:

```
>>   export(combinat):
```

To continue our exploration, we implement variations on the two previous domains, where we assume that the algebra generators are indexed by `1,2,...`. The basis elements of the free algebra and of the free commutative algebra are now respectively indexed by compositions and partitions.

```
>> domain FreeAlgebraInteger
       inherits Dom::FreeModule(Dom::Rational,
                                compositions);
       category Cat::AlgebraWithBasis(Dom::Rational);

       basisName       := hold(E);
       exprTerm        := dom::exprTermIndex;
       one             := dom::term([]);
       mult2Basis      := dom::term @ _concat;
   end_domain:
   domain FreeCommutativeAlgebraInteger
       inherits Dom::FreeModule(Dom::Rational,
                                partitions);
       category Cat::AlgebraWithBasis(Dom::Rational);

       basisName       := hold(e);
       exprTerm        := dom::exprTermIndex;
       one             := dom::term([]);
       straightenBasis := dom::term @ revert @ sort;
       mult2Basis      := dom::straightenBasis @ _concat;
   end_domain:
```

The reader may have recognized here respectively the commutative and non commutative symmetric functions, expanded on some multiplicative basis. To shorten the notations, we define two aliases, and declare the same evaluation conversion as before:

```
>> alias(NCSF = FreeAlgebraInteger,
         SF   = FreeCommutativeAlgebraInteger):
   operators::overloaded::declareConversion(NCSF, SF,
       operators::makeLinear(SF::straightenBasis,
                             Source   = NCSF,
                             ImageSet = SF)):

>> x := NCSF([1, 3, 2]);

                           E[1, 3, 2]

>> y := SF  ([1, 3, 2])

                           e[3, 2, 1]

>> SF(x)

                           e[3, 2, 1]
```

```
>> bool(SF(x)=y)

                                   TRUE
```

Let us analyze the differences with our previous implementation of the free algebras. First, we chose an indexed notation for the basis elements, as this notation is more compact and quite conventional in other systems. This is the purpose of the line `exprTerm := dom::exprTermIndex`: the method `exprTerm` of the domain is called to convert a term into an expression, as well as to print a term if there is no `printTerm` method; `dom::exprTermIndex` is a possible implementation of `exprTerm`, inherited from the category, which gives indexed notations. The other difference is that, following the usual convention, the integers in the partitions are sorted decreasingly. Here, this is suboptimally achieved by reverting the list after sorting it in the `SF::straightenBasis` method.

A disadvantage of this implementation of `SF` is that elements with many repetitions are not represented compactly:

```
>> SF([1])^10

                 e[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

One might prefer to use another basis for `SF`, where the element above would be represented as the first generator to the power 10. This can be done *via* the usual exponent notation for partitions. The basis of the algebra now consists of integer vectors. The product of two elements is simply obtained by adding up the vectors part by part, which can conveniently be implemented using the `zip` MuPAD function.

```
>> domain SFExp
       inherits Dom::FreeModule(Dom::Rational,
                                combinat::integerVectors);
       category Cat::AlgebraWithBasis(Dom::Rational);

       basisName        := hold(e);
       exprTerm         := dom::exprTermIndex;
       one              := dom::term([]);
       mult2Basis       :=
            (v1,v2) -> dom::term(zip(v1,v2,_plus,0));
   end_domain:
```

Let us do some computations:

```
>> SFExp([1]);
   SFExp([2, 0, 1])*SFExp([1, 1]);
   SFExp([1])^10
```

```
                                  e[1]

                               e[3, 1, 1]

                                  e[10]
```

This notation could be confusing, so we override it:

```
>> SFExp::exprTerm :=
       v -> _mult(dom::basisName.i^v[i] $ i=1..nops(v)):
   SFExp([1]);
   SFExp([2, 0, 1])*SFExp([1, 1]);
   SFExp([1])^10

                                   e1

                                  3
                                e1  e2 e3

                                    10
                                  e1
```

As is, the elements of this algebra are not uniquely represented. For example, the first generator of the algebra can be represented by any of `[1]`, `[1,0]`, `[1,0,0]`, `...`:

```
>> SFExp([1]), SFExp([1, 0]), SFExp([1, 0, 0]);
   bool(SFExp([1]) = SFExp([1, 0]))

                               e1, e1, e1

                                  FALSE
```

We leave it up as an exercise for the reader to fix this bug by implementing a `straightenBasis` method which strips out the trailing zeroes of the basis elements.

Of course, `SF` and `SFExp` really represent the same algebra; only the internal data representation changes. So, we provide as conversions the reciprocal isomorphisms obtained by extending by linearity the bijections `combinat::partitions::fromExp` and `combinat::partitions::toExp`:

```
>> operators::overloaded::declareConversion(SFExp, SF,
       operators::makeLinear(
           SF::term @ combinat::partitions::fromExp,
           Source = SFExp, ImageSet = SF)):
   operators::overloaded::declareConversion(SF, SFExp,
       operators::makeLinear(
           SFExp::term @ combinat::partitions::toExp,
           Source = SF, ImageSet = SFExp)):
```

Here is a simple conversion:

```
>> SF([4, 3, 3, 1]), SFExp(SF([4, 3, 3, 1]))

                                                  2
                      e[4, 3, 3, 1], e1 e3  e4
```

Let us check on an example that the conversions are indeed morphisms:

```
>> x := SF([3, 1]):
   y := SF([4, 3, 2]):
   x * y, SF( SFExp(x) * SFExp(y) )

                 e[4, 3, 3, 2, 1], e[4, 3, 3, 2, 1]
```

We can write expressions that mix elements of both domains, and let the system find a way to convert them appropriately:

```
>> ( 1 + SF([3, 1])*x ) * SFExp([2, 0]) + SFExp([1])

                              4   2      2
                            e1  e3  + e1  + e1
```

A priori, the representation of the result cannot be predicted; it depends on how the overloading mechanism chooses to resolve the conversions. If the user prefers one of the representations, she or he can take over the control at any level of the expression by forcing proper conversions:

```
>>     (1+SF([3,1])*x) * SF( SFExp([2,0]) )  +  SF( SFExp([1]) ) ;
   SF( (1+SF([3,1])*x) *     SFExp([2,0]) )  +  SF( SFExp([1]) ) ;
   SF( (1+SF([3,1])*x) *     SFExp([2,0])    +      SFExp([1]) ) ;

              e[3, 3, 1, 1, 1, 1] + e[1, 1] + e[1]

              e[3, 3, 1, 1, 1, 1] + e[1, 1] + e[1]

              e[3, 3, 1, 1, 1, 1] + e[1, 1] + e[1]
```

The implicit conversions are automatically applied transitively. As a consequence, we have without further work a conversion from `NCSF` to `SF`:

```
>> SFExp(NCSF([1, 4, 2, 2]))

                                       2
                                 e1 e2  e4
```

Another consequence is that, when there are n different representations for an algebra (say the symmetric functions expressed on any of the `p`, `e`, `m`, `h`, or `s` basis), it is sufficient *a priori* to implement n conversions to be able to get all the $n(n-1)$ possible conversions. In practice, we usually implement $2(n-1)$ conversions, so that no conversion takes more than two steps. Of course it is still possible to implement some extra direct conversions for improved efficiency; when there are several ways to convert an element from one domain to another, the system always uses one of the shortest ones.

2.2.6 A practical research example: the q-shuffle algebra

As usual, for shortening the notations, we export the library `combinat`:

```
>>   export(combinat):
```

As a more advanced example, we demonstrate some computations in the so-called q-shuffle algebra. This algebra has the sets of words for basis. The product is defined recursively by

$$\epsilon \sqcup\!\sqcup w = w \sqcup\!\sqcup \epsilon = w$$
$$ua \sqcup\!\sqcup vb := (ua \sqcup\!\sqcup v)b + q^{|vb|}(u \sqcup\!\sqcup vb)a.$$

where ϵ is the empty word, u, v, w are words, a, b letters and $|w|$ denotes the length of the word w.

```
>> domain qShuffleAlgebra(K: DOM_DOMAIN)
       // The parameter K is the base field

       // Implementation of the vector space structure
       inherits Dom::FreeModule(K, words);

       // This is an algebra with a distinguished basis
       category Cat::AlgebraWithBasis(K);

       // Basis elements are printed as W([a,b,a,a])
       basisName     := hold(W);

       // The unit of the algebra is the empty word
       one           := dom::term([]);

       // The binary product of the algebra is defined
       // by linearity on basis elements
       mult2Basis    :=
       proc(ua: dom::basisIndices, vb: dom::basisIndices)
           local a, b, u, v;
```

```
    begin
    // Base case: ua is empty
        if nops(ua)=0 then return(dom::term(vb)); end_if;
    // Base case: vb is empty
        if nops(vb)=0 then return(dom::term(ua)); end_if;
    // extract u and a from ua
        u := [op(ua, 1..nops(ua)-1)]; a := ua[nops(ua)];
    // extract v and b from vb
        v := [op(vb, 1..nops(vb)-1)]; b := vb[nops(vb)];
    // the recursion formula
        dom::mapterms(dom::mult2Basis(ua, v), append, b) +
        dom::mapterms(dom::mult2Basis(u, vb), append, a) *
           q^nops(vb)
    end_proc;
end_domain:
```

Note: in a free module, the entry `dom::basisIndices` contains the combinatorial class that indexes the basis of the module, and the function `mapterms(elts, f, ...)` applies `f` on each term of the element `elts` of the module. Thus, if `elts` writes $\sum_w c_w\, w$, the call

```
    dom::mapterms(elts, append, b)
```

returns the MuPAD representation of $\sum_w c_w\, wb$.

Let us declare this algebra over the field of expressions:

```
>> W := qShuffleAlgebra(Dom::ExpressionField()):
```

and try some computations:

```
>> W([a])*W([b]);
   W([b])*W([a]);

                    q W([b, a]) + W([a, b])

                    W([b, a]) + q W([a, b])
```

Obviously, this is a non commutative algebra. One can use any object as letter:

```
>> W([b, c, d])*W([2, 3])

  3                         4
 q  W([b, 2, c, 3, d]) + q  W([2, b, c, 3, d]) +


    2                         6
   q  W([b, c, 2, 3, d]) + q  W([2, 3, b, c, d]) +
    4                         5
   q  W([b, 2, 3, c, d]) + q  W([2, b, 3, c, d]) +

                           2
   W([b, c, d, 2, 3]) + q  W([b, 2, c, d, 3]) +

    3
   q  W([2, b, c, d, 3]) + q W([b, c, 2, d, 3])
```

Moreover, the automatic conversion from the base field allows one to write complicated expression such as:

```
>> 1/2*W([b])^3 + W([a])*(1 + W([1, 2]))

  2
 q  W([1, 2, a]) + W([a, 1, 2]) + q W([1, a, 2]) +

    /               2      \
    | (q + 1) (q + q  + 1) |
    | -------------------- | W([b, b, b]) + W([a])
    \           2          /
```

Finally note that this implementation is not very efficient. In particular, at each stage of the recursion in the expression:

```
   dom::mapterms(dom::mult2Basis(ua, v ), append, b) +
   dom::mapterms(dom::mult2Basis(u,  vb), append, a) * q^nops(vb)
```

the system has to solve the overloading, i.e., to decide the meaning of `+` and `*`. It is better to decide this once and for all, i.e., to replace `+` by `dom::plus2`, and `*` by `dom::multcoeffs`. The emerging expression is less readable, but its processing is considerably faster:

```
   dom::plus2(dom::mapterms(dom::mult2Basis(ua, v ),
                            append, b),
        dom::multcoeffs(
             dom::mapterms(dom::mult2Basis(u,  vb),
                            append, a),
                        q^nops(vb)));
```

Moreover, a non-recursive implementation of the q-shuffle is likely to be more efficient.

2.2.7 Sample applications

We present here some typical applications that involve simultaneously the combinatorial and the algebraic tools of MuPAD-Combinat together with the general computer algebra tools of MuPAD.

First, we find the minimal polynomial of an element of the group algebra of the symmetric group of order 4:

```
>> export(combinat):
   domain AlgSymGroup4
       inherits Dom::FreeModule(
                   Dom::ExpressionField(),
                   subClass(permutations, Parameters = 4));
       category Cat::AlgebraWithBasis(Dom::ExpressionField());

       basisName       := hold(p);
       exprTerm        := dom::exprTermIndex;
       one             := dom::term([1,2,3,4]);
       mult2Basis      := dom::term @ permutations::mult2;
   end_domain:
   x := AlgSymGroup4([2,3,4,1]):
   y := x^0 + a1*x^1 + a2*x^2 + a3*x^3 + a4*x^4;

 a3 p[4, 1, 2, 3] + a2 p[3, 4, 1, 2] + a1 p[2, 3, 4, 1] +

    (a4 + 1) p[1, 2, 3, 4]

>> solve([coeff(y)], [a1,a2,a3,a4]);

              {[a1 = 0, a2 = 0, a3 = 0, a4 = -1]}

>> subs(z^0 + a1*z^1 + a2*z^2 + a3*z^3 + a4*z^4, op(last(1),1))

                                     4
                                1 - z
```

Second, we use Pólya enumeration theory to count unlabelled non oriented graphs on n nodes without loops. To this end, we build the symmetric group S_n seen as the group of the permutations of the n nodes $\{i\}$ of the graphs, and the induced permutation group G_n on the set E of the $\binom{n}{2}$ possible edges $\{i, j\}$ between those nodes:

```
>> S4 := Dom::SymmetricGroup(4):
   G4 := Dom::PermutationGroupOnSets(S4, 2):
```

We compute next the cycle indicator of G_4. This is a symmetric function which encodes the statistic of the cycle types of the permutations in G_4:

```
>> Z4 := G4::cycleIndicator()

 1/24 p[1, 1, 1, 1, 1, 1] + 3/8 p[2, 2, 1, 1] + 1/3 p[3, 3] +

    1/4 p[4, 2]
```

This is to be interpreted as follows: in G_4, there are $6 = |G_4| \cdot \frac{1}{4}$ permutations with one cycle of length 4 and one cycle of length 2. The interesting fact about this symmetric function is that, when evaluated on an alphabet $A = (a_1, \dots, a_n)$, it returns the generating series by weight for the functions from E to a set of size n whose elements are weighted by A. For example, a graph can be seen as a function from E to a set with 2 elements. Hence, the total number of unlabelled graphs is given by:

```
>> Z4([1, 1])

                                    11
```

whereas the generating series of unlabelled graphs on 4 nodes by number of edges is:

```
>> expand(Z4([1, q]))

                     2       3       4     5     6
              q + 2 q  + 3 q  + 2 q  + q  + q  + 1
```

Such computations are carried out in a rather efficient way, so that e.g. counting the number of graphs with 20 nodes takes just a few seconds:

```
>> (Dom::PermutationGroupOnSets(Dom::SymmetricGroup(20),
                                 2))::cycleIndicator()([1, 1])

             645490122795799841856164638490742749440
```

If one is instead interested in counting multigraphs (graphs with multiple edges) by number of edges, the cycle indicator polynomial can be evaluated on the infinite alphabet $A := (1, q, q^2, \dots)$. Infinite alphabets are not yet directly supported by MuPAD-Combinat; however this can easily be done by hand since the evaluation of the symmetric powersum p_k on the alphabet A is obtained by encoding A as $1 + q + q^2 + \dots = \frac{1}{1-q}$ and substituting q by q^k in this formula:

```
>> H4 := _plus(_mult(op(term, 1),
                     1/(1 - q^k) $ k in op(term, 2))
               $ term in poly2list(Z4))
```

```
          3                  1                  1
-------------------- + ----------- + ------------------ +
    2     2        2              6       2        4
8 (q  - 1)  (q - 1)    24 (q - 1)    4 (q  - 1) (q  - 1)

        1
   -----------
       3     2
   3 (q  - 1)
```

Now, the number of multigraphs with 0 to 4 edges can be obtained by Taylor expansion:

```
>> series(H4, q, 5)

                          2      3       4      5
              1 + q + 3 q  + 6 q  + 11 q  + O(q )
```

In fact, this method is used to implement counting in the combinatorial class `combinat::integerVectorsModPermutationGroup`:

```
>> M4:=combinat::integerVectorsModPermutationGroup(G4):

>> M4::generatingSeries(q)

          3                  1                  1
-------------------- + ----------- + ------------------ +
    2     2        2              6       2        4
8 (q  - 1)  (q - 1)    24 (q - 1)    4 (q  - 1) (q  - 1)

        1
   -----------
       3     2
   3 (q  - 1)

>> M4::count(i) $ i = 0..10

            1, 1, 3, 6, 11, 18, 32, 48, 75, 111, 160
```

2.2.8 Advanced algebraic structures

The current development aims to provide a framework for advanced algebraic structures, such as modules with several bases or Hopf algebras, with plans for Lie algebras and operads in the long run. In this section, we demonstrate how to implement such structures. At the time of writing, the user interface is not fully stabilized, so please check the latest available documentation before trying out the examples.

We start with an example of algebra with several bases. Let S be a finite set, and consider the free module M over the subsets of S. It is endowed with an algebra structure by extending the intersection operation by linearity. Implementing this algebra in MuPAD requires some helper tools; we put them inside a dummy domain called `SubsetsSpaceTools` to avoid polluting the global name space:

```
>> domain SubsetsSpaceTools
       info := "Helper tools for the domain 'SubsetsSpace'";
   end_domain:
```

We now implement the module M with elements expanded on the fundamental basis denoted by F. There are two parameters: the set `S` and, as usual, the coefficient ring `Ring`. Note the use of `combinat::subClass` to define the combinatorial class of the subsets of `S`:

```
>> domain SubsetsSpaceTools::Fundamental(S, Ring)
       category Cat::AlgebraWithBasis(Ring);
       inherits Dom::FreeModule(Ring,
                    combinat::subClass(combinat::subsets,
                                       Parameters = S));
       info := "The subset space on the fundamental basis";
       basisName := hold(F);

       mult2Basis :=
       proc(s1: dom::basisIndices, s2: dom::basisIndices)
       begin
           dom::term(s1 intersect s2);
       end_proc;
   end_domain:
```

Let us just recall that, in a free module, the entry `dom::basisIndices` contains the combinatorial class that indexes the bases of the module. Here, this is a shortcut for the combinatorial class:

```
   combinat::subClass(combinat::subsets, Parameters = S)
```

The module M has two other bases In and Out that we describe below. We just declare them to MuPAD, without implementing the product:

```
>> domain SubsetsSpaceTools::In(S, Ring)
       category Cat::AlgebraWithBasis(Ring);
       inherits Dom::FreeModule(Ring,
                    combinat::subClass(combinat::subsets,
                                       Parameters = S));
```

```
    info := "The subset space on the 'In' basis";
    basisName := hold(In);
end_domain:

domain SubsetsSpaceTools::Out(S, Ring)
    category Cat::AlgebraWithBasis(Ring);
    inherits Dom::FreeModule(Ring,
                 combinat::subClass(combinat::subsets,
                                    Parameters = S));
    info := "The subset space on the 'Out' basis";
    basisName := hold(Out);
end_domain:
```

The module M is endowed with a scalar product which makes the fundamental basis F orthonormal. The In basis is then defined by the rule

$$\mathrm{In}_S = \sum_{X \subset S} F_X,$$

and the Out basis is the dual basis of In with respect to the scalar product. This information is sufficient to define mathematically all the bases change in between F, In, and Out by transposing and inverting matrices. The implementation follows the same compact line:

```
>> domain SubsetsSpace(S : DOM_SET,
             Ring = Dom::Rational : DOM_DOMAIN)
       category Cat::ModuleWithSeveralBases(Ring);
       inherits Dom::BaseDomain;

       info := "The subsets space on various bases";

       Fundamental := SubsetsSpaceTools::Fundamental(S, Ring);
       F           := dom::Fundamental; // a shortcut
       In          := SubsetsSpaceTools::In(S, Ring);
       Out         := SubsetsSpaceTools::Out(S, Ring);

       // When possible, define automatically basis changes by
       // transposition or inversion of matrices.
       autoDefineBasisChanges := TRUE;

       basisChangesBasis := table(
           (dom::In, dom::F) =
               proc(set : dom::In::basisIndices) : dom::F
                   local xSet;
```

```
                begin
                    dom::F::plus(dom::F::term(xSet) $
                        xSet in combinat::subsets::list(set));
                end_proc);

        dual := dom; // The dual of this space.
    // The pairs of dual bases.
    // The pair [dom::Out, dom::In] will be automatically declared.
        dualBasesPairs := {[dom::F, dom::F], [dom::In, dom::Out]};

    end_domain:
```

The crucial line of the preceding code is the declaration of `SubsetsSpace` as a `Cat::ModuleWithSeveralBases`. This category provides helper tools for implementing modules which are represented on several modules. We just need to supply the basis changes from `In` to `F` (see `basisChangesBasis`) and to declare which bases are in duality (see `dual` and `basisChangesBasis`). The system then provides default implementations for the scalar products and the other changes of bases (see `autoDefineBasisChanges`), and provides the standard free module entries `testtype` and `coeffRing` together with the set of bases `bases`.

We define the free module spanned by the subsets of $\{1, 2, 3, 4\}$ and build some of its elements:

```
>> M1234 := SubsetsSpace({1, 2, 3, 4}):
   el1   := M1234::F({1, 2});
   el2   := M1234::In({1, 2})

                                F({1, 2})

                               In({1, 2})
```

The type checking works as usual:

```
>> testtype(el1, M1234), testtype(el2, M1234)

                               TRUE, TRUE
```

as well as the bases changes:

```
>> M1234::In(el1)

            - In({2}) - In({1}) + In({}) + In({1, 2})

>> M1234::Out(el2)
```

```
 4 Out({1, 2, 3, 4}) + 2 Out({2, 3, 4}) + 2 Out({1, 3, 4}) +

    4 Out({1, 2, 4}) + 4 Out({1, 2, 3}) + Out({3, 4}) +

    2 Out({2, 4}) + 2 Out({2, 3}) + 2 Out({1, 4}) +

    2 Out({1, 3}) + Out({4}) + Out({3}) + 2 Out({2}) +

    2 Out({1}) + Out({}) + 4 Out({1, 2})
```

The required matrix inversions are computed only once and remembered, so that later basis changes are considerably faster, at some memory cost.

Finally we check that algebra products and scalar products are handled correctly:

```
>> el1*el2

               F({2}) + F({1}) + F({}) + F({1, 2})

>> operators::scalar(el1, el2)

                                 1
```

Products of `Out` elements are actually computed and returned in the `F` basis. As usual, an explicit conversion can be used to force the result back in the `Out` basis:

```
>> M1234::Out({1, 3, 4})*M1234::Out({2, 3});

 - F({1, 2, 3, 4}) + F({2, 3, 4}) + F({1, 3, 4}) +

    F({1, 2, 3}) - F({3, 4}) - F({2, 3}) - F({1, 3}) + F({3})

>> M1234::Out(last(1))

                             Out({3})
```

Let us finish this tour by showing a small example of Hopf algebra: the free algebra over the integers, those being primitive for the coproduct (non commutative symmetric functions on one of the powersum bases):

```
>> domain FreeAlgebraInteger
       inherits Dom::FreeModule(Dom::Rational,
                                combinat::compositions);
       category Cat::GradedHopfAlgebraWithBasis(Dom::Rational);

       basisName       := hold(P);
       exprTerm        := dom::exprTermIndex;
```

```
        one             := dom::term([]);
        mult2Basis      := dom::term @ _concat;
        coproductBasis  :=
        proc(compo : dom::basisIndices)
            local i, tens;
        begin
            tens := dom::tensorSquare;
            tens::mult(tens::plus2(tens::term([[i], []]),
                                   tens::term([[],  [i]]))
                       $ i in compo);
        end_proc;
    end_domain:
    alias(NCSF = FreeAlgebraInteger)
```

Here is a sample coproduct computation:

```
>> tens := operators::coproduct(NCSF([2, 1, 3]))

 tensor(P[2, 3], P[1]) + tensor(P[1, 3], P[2]) +

    tensor(P[2, 1], P[3]) + tensor(P[3], P[2, 1]) +

    tensor(P[1], P[2, 3]) + tensor(P[2], P[1, 3]) +

    tensor(P[2, 1, 3], P[]) + tensor(P[], P[2, 1, 3])
```

To improve the readability, we use ø (iso-latin-1 character of code 248) as symbol for the tensor product:

```
>> operators::setTensorSymbol("ø"):
   tens

 P[2, 3] ø P[1] + P[1, 3] ø P[2] + P[2, 1] ø P[3] +

    P[3] ø P[2, 1] + P[1] ø P[2, 3] + P[2] ø P[1, 3] +

    P[2, 1, 3] ø P[] + P[] ø P[2, 1, 3]
```

ø can be use as well to type in tensor products:

```
>> NCSF([1,2]) ø NCSF([1])

                        P[1, 2] ø P[1]
```

The product of NCSF is naturally extended to NCSF $\otimes$ NCSF, so that we can compute expressions such as:

```
>>   tens * (NCSF([1,2]) ⊗ NCSF([1]))

 P[2, 1, 3, 1, 2] ⊗ P[1] + P[2, 1, 2] ⊗ P[1, 3, 1] +

    P[1, 1, 2] ⊗ P[2, 3, 1] + P[3, 1, 2] ⊗ P[2, 1, 1] +

    P[2, 1, 1, 2] ⊗ P[3, 1] + P[1, 3, 1, 2] ⊗ P[2, 1] +

    P[2, 3, 1, 2] ⊗ P[1, 1] + P[1, 2] ⊗ P[2, 1, 3, 1]
```

Note that a new domain representing the tensor square has been automatically created:

```
>> domtype(tens)

          Dom::TensorProductOfFreeModules([NCSF, NCSF])
```

Since NCSF is a graded connected bi-algebra, it is automatically a Hopf algebra, whose antipode can be computed recursively (in a slow way):

```
>> operators::antipode(NCSF([2, 1, 3]))

                         (-1) P[3, 1, 2]
```

Of course, in the case of NCSF, it would have been much more efficient to implement the direct combinatorial formula for the antipode:

```
     antipodeBasis :=
         comp -> dom::monomial(-1^nops(comp), revert(comp));
```

Another very useful feature is the possibility to define tensor product of maps:

```
>> idTensorAntipode := operators::tensorProductOfMaps(
       [id, NCSF::antipode],
       NCSF::tensorSquare,
       NCSF::tensorSquare):
   idTensorAntipode(tens)

 - P[2, 3] ⊗ P[1] - P[1, 3] ⊗ P[2] - P[2, 1] ⊗ P[3] +

    P[3] ⊗ P[1, 2] + P[1] ⊗ P[3, 2] + P[2] ⊗ P[3, 1] +

    P[2, 1, 3] ⊗ P[] - P[] ⊗ P[3, 1, 2]
```

We conclude by using this feature to check, on some examples, that the antipode is correct:

```
>> es := NCSF::mu @ idTensorAntipode @ NCSF::coproduct:
```

```
>> es(NCSF([2, 1]));
   es(NCSF([]));

                                        0

                                       P[]
```

2.3 Current features

We conclude this guided tour by a summary of the current features. A first part of the package consists of predefined libraries to count, enumerate, and manipulate standard combinatorial objects (partitions, compositions, sets, words, permutations, tableaux, trees, . . .), together with generic tools to help define new combinatorial classes:

- A computational engine for dealing with integer vectors with prescribed constraints (monomials, compositions, partitions, Dyck paths, . . .)
- A computational engine for generating linear extensions of posets (standard young tableaux, standard ribbons, . . .)
- A refactored and extended version of the former `CS` library by S. Corteel, A. Denise, I. Dutour, and P. Zimmermann to deal with objects defined by a recursive grammar.

Most predefined libraries actually make use of these engines.

A second part consists of tools to build combinatorial algebras. The typical usage is to take a vector space with basis indexed by some combinatorial objects, to define a product for two basis elements and to extend the product by bilinearity. The system automatically takes care of the data structure of algebraic elements, of extensions of functions by linearity, bi-linearity, or associativity, of conversions between different bases, etc. Similarly, one may define coproducts, antipodes, and similar operators, to implement more involved structures such as Hopf algebras. Some preliminary work has been done to manipulate Lie algebras and operads as well. In short, the system takes care of the algebraic bookkeeping, so that one can concentrate on the underlying combinatorics rather than on the programming.

As examples of usage and applications, we provide a library for the algebra of symmetric functions, and we have (currently undocumented) libraries for the algebra of non commutative symmetric functions, the algebra of (free) quasi-symmetric functions, the Loday-Ronco algebra, the Weyl algebra, the rational Steenrod algebra, the type-A Hecke and Hecke-Clifford algebras, as well as invariant rings of permutation groups, and group algebras.

In the future we plan to provide predefined libraries for the free symmetric algebra, the algebra of matrix quasi symmetric functions, the descents and peaks algebras, general Ore-algebras, the symmetric Weyl algebra, the algebra of multi-symmetric functions, the divided power algebra, free Lie algebras, etc.

Finally, we provide libraries for manipulating weighted automatons and related (tropical) semi-rings.

3 The design of the MuPAD-Combinat package

3.1 The development platform

The choice of the development platform was a difficult question; at some point, after long discussions, we had to take a decision. We try to present here why we were led to choose MuPAD. The table 1 summarizes how we evaluated several platforms with respect to our specifications for this project (see the specifications). This evaluation is based on our personal daily experience, in our field, with our programming and computing habits; we have tried to be as objective as possible, but we do not claim any kind of scientific rigor nor universality. We will also be glad to update it, if some readers explain to us why their favorite system has been underrated.

A short summary is in order. The development cycle is too long in programming languages like C++, and there are not yet strong enough computer algebra libraries for our needs. We would have preferred to use a computer algebra system that was already widely used, in particular in the algebraic combinatorics community, like the commercial system Maple [?] or Mathematica [?]. However, the programming language, the license conditions, and the long term maintenance of Maple are terrible. Technically, Mathematica looks better, but does not seem to fit our technical requirements concerning object oriented features and overloading. Not going for the academic system GAP [?] was a tough choice, as they have an excellent open source development model and a clean programming language. What kept us from choosing it is that general computer algebra tools are missing. Still we want to collaborate closely with the GAP team, for example for using GAP as a group theory computation server. Axiom [?] was not yet open-source at that time, and even today, its future seems still to be uncertain; aside from this, its language and design, together with the existence of the Aldor [?] compiler, makes it an excellent candidate; in fact, many ideas in the design of MuPAD, like domains and categories, are borrowed from Axiom. Aldor is a computer algebra system in its own, but the computer algebra libraries are in construction. Again, its type system makes Aldor as one of the most interesting systems for our goals. Finally, Magma [?, ?] was quite appealing, in particular for its impressive efficiency. However, we are afraid that their closed development model, with a lot of the work being done in C in the kernel, will not scale in the long run. Also it does not offer us the programming flexibility that we need.

Only the future will tell us if our choice of MuPAD was a right one. So far, after three years of development, the programming language has proved to fit well our needs, and the collaboration with the MuPAD group has worked out beautifully. Just to give a figure, in 2003, 400 out of the 1400 messages on the mupad-combinat-devel mailing list originated from the MuPAD team; this included code reviews, explanations on the internals of MuPAD, tips and tricks for optimizations, discussions on common developments that deserved to

Aspect	Relevance	`Caml`	`C++`	`GAP`	`Java`	`Magma`	`Maple`	MuPAD	`Mathematica`	`Maxima`	`Axiom`	`Aldor`
License	B	A	A	A	A	C	E	B	E	A	A	A
Community	C	B	A	C	A	C	B	C	B	D	D	D
Potential user base	B	C	C	B	C	B	A	B	A	C	C	C
Long term availability	B	A	A	B	A	B	A	B	A	?	?	?
Long term compatibility	A	A	C	?	A	B	E	A	B	?	C	C
Support	A	A	A	A	A	B	D	A	B	?	D	D
Development cycle	A	B	E	?	?	?	C	B	?	?	D	D
Ease of use for beginners	B	C	D	B	C	B	A	A	A	?	C	C
Functional programming	A	A	C	B	C	?	D	B	?	?	C	C
Typing/Object Oriented	A	A	A	B	A	C	E	B	?	F	A	A
Overloading	B	?	A	A	D	?	F	B	?	?	A	A
Term rewriting	D	?	?	?	?	?	?	C	A	?	?	?
Interactive usage	B	C	C	A	C	A	A	A	A	A	A	A
Compilation	C	A	A	A	A	B	F	F	?	?	B	A
Programming tools	A	A	A	B	A	?	D	B	?	?	B	A
Computer algebra tools	A	C	C	C	D	B	B	B	?	?	B	C
Linear algebra tools	B	?	B	B	?	B	B	C	?	?	C	?
Group theory tools	C	?	?	A	?	A	E	E	?	?	C	C
Inter Process Com.	B	A	A	B	A	?	C	C	?	?	D	C
Dynamic modules	B	A	A	A	A	F	B	B	?	?	D	A
Overall		C	C	B	D	C	D	B	?	?	B	B

Table 1: Rating of some platforms we investigated with respect to different aspects we judged of importance. Each rating ranges from F (worst) to A (best), ? meaning a lack of information. For example, `Caml` is a full featured functional programing language and is rated A, whereas MuPAD does not support at all compilation of user code, and is rated F. For general purpose programming language, we tried to take into account open source or freely available libraries to build upon. This includes for example `FOC` for `Caml`, or `SYNAPS` and `LinBox` for `C++`.

be synchronized, and so on.

Having our code deeply integrated in the MuPAD library is of great benefit for us. First, we gain in visibility, since our code is widely distributed. Second, it enforces high standards for code stability and documentation; though this is quite time consuming for us and imposes deadlines which are not necessarily compatible with our research schedule, this is essential for attracting new users. Also, our code and documentation are tested every night on the MuPAD servers as part of their standard test-suite. In this way, any incompatibilities are instantly detected. In particular, the MuPAD developers have to take our project into account before deciding any backward incompatible change that could break our code; if the change is still done, they are forced to either fix our code directly, or at least to provide us with timely information on how to do it ourselves. This is essential for the long term maintenance of our project. Reciprocally, some of our suggestions have had an essential impact in their decision for several small but critical kernel changes; this helped us to write cleaner and more efficient code.

Altogether, there is essentially one single aspect we are not happy with for the long term survival of `MuPAD-Combinat`, namely that MuPAD is not open-source. Still, they have kept so far their promises to remain relatively open. The situation has even improved recently with MuPAD Light being completely free (gratis) for research and education. They also have promised to release the code source of the library under a well known open-source license, some day.

3.2 The development model

Looking back at previous projects shows that the choice of a proper development model is essential for the long term survival of a project. In fact, the first months of the development of this project have been spent in large parts discussing this issue and trying out different solutions.

First, it is important that the *core* development be done by several persons with permanent positions to avoid the left-for-industry dreaded effect. Of course, this does not preclude students from writing libraries around this core, on the contrary! Also, code sharing code is time-consuming: seen over a short period, providing a polished implementation of a routine, with documentation and tests, represents much more work than just going for a write-once-use-once implementation. There, it is essential to reach quickly the critical mass of developers so that altogether they actually save time by participating to the project and benefiting from other's code.

In this spirit of sharing code and attracting new users and developers, going for an open source license was an obvious choice. Having an open platform would have helped as well. What is less obvious is how much the standard open source development models apply to the algebraic combinatorics community. The main problem is that this community is relatively small with not so many potential participants. The fact that participating in such a software requires the use of several standard open source management tools (cvs, make, Wiki) which are essential to automatize things may be seen as an additional difficulty. We try our best to leverage those difficulties, and the support center is of great help for that. But still, some of those tools have a steep learning curve. Also, the structure of the package, which as been designed with genericity and extensibility in mind, is clear and systematic, but not so simple, and relies on non trivial programming concepts (generators, inheritance, ...).

As a consequence of this, and despite much documentation, advertising, and teaching efforts, we have not been as successful as we would have desired at attracting new users and developers. In particular, we have not yet quite reached the required critical mass. Another aspect of this is that the project is not yet as international and lab-independent as it ought to be to ensure its political independence. It seems that, so far, getting new developers started still requires a regular physical contact.

3.3 Naming conventions

Long names versus abbreviations

The convention for library, domain and function names is to use long names that are as meaningful and close to the English spelling as possible: e.g. `partitions` instead of `PART`, `FreeQuasiSymmetricFunctions` instead of `FQSym`, etc. In particular, abbreviations should be avoided, except in extreme cases where the short name is really well established (say Lex instead of Lexicographic). Here are the motivations for this convention:

- This is the convention used in MuPAD;
- Since MuPAD $\geq$ 2.0.0 has automatic name completion, long names are not too much of a pain to type in.
- This is helpful for users coming from other areas;
- The user can easily define shorthands (via aliases or assignments) for the functions he uses a lot. Actually this is quite reasonable: a working session starts by the definition of the notations and shorthands, exactly as any mathematical document. Tip: in our daily usage, we have one file per topic we do research on, with a set of appropriate shorthands. Typically, when working on the Loday-Ronco Algebra, we use shortcut `BT` for `combinat::binaryTrees`, `Perm` for `combinat::permutations`, `LRA` for the Loday-Ronco Algebra, and so on.
- Given the variety of areas that intersect on combinatorial algebras, there are too many risks of conflicts with short names; the user needs to be able

to choose his own notations given the context and the set of objects that are to be manipulated at the same time.

Case of names

We follow the capitalization rules of the MuPAD coding standard, which are quite similar to `Java` or `C++` rules:

- When a name is composed of several parts, the later parts are separated by capitalizing the first letter of the following parts. For example, "from reduced word" yields `fromReducedWord`. Using underscores to separate parts (as in `from_reduced_word`) is not recommended; some names in our code do not follow this recommendation yet.
- Names of options and local variables of domains are capitalized (`MinLength`, `R`).
- Names of normal variables, of functions, and of methods are not capitalized (for example `combinat::partitions::type`, `Dom::Matrix::transpose`, `combinat::permutations::fromReducedWord`).
- A few internal variables are fully capitalized to alert the user that they have a very specific behavior, and should be used with care (`DOM`, `TEXTWIDTH`).
- Badly enough neither the `MuPAD-Combinat` package, nor the MuPAD standard library do respect any clearly defined rule for domain names. As a rule of thumb, the name of a domain is not capitalized when the domain is a library (`combinat`, `combinat::generators`, and is capitalized when it is a true domain which contains elements (`Series`, `Dom::Rational`, `examples::SymmetricFunctions`). The later case includes for example all the domains in `Dom`, `examples`, `experimental`. On the other hand, the names of combinatorial classes in `combinat` are not (yet) capitalized (`combinat::partitions`). Other exceptions to this rule typically appear when the name of a library comes from initials (`IPC`) or from a person name.

  This lack of coherency is a burden for both users and developers, and we hope to fix it at some point in the future, when the MuPAD library will undergo a similar naming convention cleanup.

Composite names

When the name of a library, domain or function is composed of several parts, and those parts are also used in other names, it may be worthwhile to order those parts from the most general to the most specific. For example, we used the name `combinat::integerVectorsWeighted` instead of the more natural name `combinat::weightedIntegerVectors`. The advantage is that all the domains dealing with integer vectors start with the same prefix, which is particularly practical with respect to automatic completion. This is also coherent with the hierarchy `library::sublibrary::subsublibrary`. Another typical case is when several functions return a similar result but under different forms; the function that is the most useful or natural gets the short name, and the names of the other functions, are suffixed with the "type" of the result (`words::inversions`/`words::inversionsList`, ...). We also used this rule of thumb for the names of the free module methods

- `mult`/`multBasis`/`mult2`/`mult2Basis`,
- `straighten`/`straightenBasis`,
- `print`/`printTerm`/`printMonomial`/`printBasis`.

3.4 Representing combinatorial objects and classes

The notion of *combinatorial object* is best described by some examples: a partition, a binary tree, a permutation, a graph, a Feynman diagram, a Dyck word, and other similar discrete objects are all combinatorial objects.

A *combinatorial class* is a (countable) set of related combinatorial objects (e.g. the set of all partitions, the set of all binary trees, the set of all standard permutations), on which a *size* function is defined (e.g. the size of partition is the sum n of its parts; the size of an integer vector consist of a pair (n, k) of integers: its sum n and its length k; the size of a tree is the number of its nodes). Typically, the fibers of the size function define natural finite subsets of the class that one wants to count, enumerate, and so on (e.g. counting all the partitions of $n = 4$, listing all the integer vectors of sum 5 and length 3); we say "typically", because there are some cases where it is practical to use this framework even if the subsets are only countable. In many cases optional restrictions can be added to define smaller subsets of the class to be counted/enumerated/..., (e.g. the partitions of 4 of length at most 4). In some combinatorial classes (e.g. the class of the permutations of 5), the size function may be degenerated and have only one non trivial fiber.

Representing combinatorial objects

A combinatorial object may belong to several combinatorial classes simultaneously. For example, the list `[4,3,2,1]` may be interpreted as a partition, a permutation, an integer vector, a composition. This is reflected in MuPAD by our convention that a combinatorial object is not necessarily strongly typed by the combinatorial class(es) to which it belongs. An object has a unique domain: it corresponds to the data structure of the object and can be obtained by the command `domtype`. On the other hand, it may be of different types: they correspond to the different semantics that can be attached to the object, and they can be tested with `testtype`.

For example, `[3,4,2,1]` belongs to the MuPAD domain of lists, `DOM_LIST`; it is simultaneously a list of positive integers (`Type::ListOf(Type::PosInt)`), a word, a permutation, etc, while it is not a partition:

```
>> domtype ([3, 4, 2, 1]);

                              DOM_LIST

>> testtype([3, 4, 2, 1], Type::ListOf(Type::PosInt)),
   testtype([3, 4, 2, 1], combinat::words),
   testtype([3, 4, 2, 1], combinat::compositions),
   testtype([3, 4, 2, 1], combinat::integerVectors),
   testtype([3, 4, 2, 1], combinat::permutations),
   testtype([3, 4, 2, 1], combinat::partitions)

                 TRUE, TRUE, TRUE, TRUE, TRUE, FALSE
```

In the MuPAD terminology, the domains like `combinat::compositions` are called *facade domains*; they do not really contain elements of their own.

On the other hand, some combinatorial objects, such as trees, require a specific data structure; these objects are strongly typed, that is their domain is the class itself. This has, among others, the advantage that they are pretty printed by the system:

```
>> t := combinat::labelledBinaryTrees([1, [2], [3]]);
   domtype(t);
   testtype(t, combinat::labelledBinaryTrees);

                                  1
                                 / \
                                 2 3

                     combinat::labelledBinaryTrees

                                 TRUE
```

This choice of not systematically using strong typing for combinatorial classes is not an obvious one, and there is no clear cut criteria for deciding whether a given combinatorial class should use strong typing or not. On the one hand, strong typing allows for object-oriented techniques and overloading. On the other hand, having to cope with all the conversions (a partition is also a composition, ...) can be quite a burden for both the user and the programmer; indeed, choosing which implicit conversions to provide is not an easy task, given the overwhelming number of natural bijections. Finally, with the current MuPAD language, there is a small memory and time overhead with strong typing; this can be considered as a misfeature of MuPAD though.

Aside from the data structure criteria, another reasonable criteria is whether the combinatorial class has natural “algebraic operations”. This is why we currently have both a facade domain `combinat::permutations` for general permutations seen as words, and a real domain `Dom::SymmetricGroup` for standard permutations seen as elements of the symmetric group. Actually, it could be reasonable to have a real domain `Dom::Permutation`, and have `Dom::SymmetricGroup` and `Dom::PermutationGroup` be facade domains for `Dom::Permutation`. This is still under discussion, and comments are welcome.

Representing combinatorial classes

Combinatorial classes for which we want to do counting, generation, or other manipulations are represented by MuPAD domains, like `combinat::partitions` or `combinat::binaryTrees`. Note that, in many cases, those domains are just *facade domains* and do not really contain elements: as we said above, the domain of a partition, or of a permutation, is really `DOM_LIST`. Those domains also define a MuPAD type; by convention, it is named like `combinat::partitions::type`, and can be tested with:

```
>> testtype([3, 2, 2, 1], combinat::partitions)

                              TRUE
```

Simpler combinatorial classes, which we only want to use for type checking, are just represented by MuPAD types. This is typically used for subsets of other combinatorial classes. For example, the standard permutations form a subset of all permutations, and are represented by the type `combinat::permutations::typeStandard`. We are not quite happy with this naming convention; however, for better localization, we really would like to keep the types defining a subset of a domain inside this domain. Another option was to use subdomains even in this case. But domains are quite special (and expensive?) objects in MuPAD: they have a reference effect, they cannot be deleted, etc. So, we feel that this would be overkill, especially for parametrized types like `PermutationOf([a,b,c,d])`.

Another related situation: quite often, we have a function that returns a collection C of related objects, usually as a list. Think of `combinat::words::shuffle([1,2,3],[a,b,c])` which returns a list of words. Or think of the inverse of a function that is not at all injective like `combinat::permutations::fromCycleType` (it returns all the permutations having a given cycle type). In such cases, we often want to do some more involved things, like having a generator for the elements of C, or being able to count them without actually generating them. Then, it is natural to consider C as a combinatorial class, and to represent it by a MuPAD domain. This gives a unified interface to all the standard functions for counting, generating, ...:

- `combinat::words::shuffle::count(word1,word2)`
- `combinat::words::shuffle::list(word1,word2)`
- `combinat::words::shuffle::generator(word1,word2)`

As a nice side effect, the standard alias from `new` to `list` allows one to use the natural syntax `combinat::words::shuffle(word1,word2)` to obtain the collection C. So, switching from a simple function which returns C to a domain for the elements of C is transparent for the user. Usually, such a domain will be a subdomain of an existing domain (here `combinat::words::shuffle` is a subdomain of `combinat::words`).

Combinatorial classes and categories

A domain which represents a combinatorial class belongs to the category `Cat::CombinatorialClass`. Such a domain should implement at least `generator` or `list`. This category also provides naming conventions for usual functions like `count`, `first`, `next`, `random`, `unrank`, ... The implementation of those functions is not explicitly required by the category `Cat::CombinatorialClass`: depending on the specific combinatorial class sometimes they are not yet implemented, sometimes there exists no efficient algorithm, or sometimes they simply do not really make sense.

In the future, we may think about refining `Cat::CombinatorialClass` into subcategories that describe which of those functions are available (for example, the category of combinatorial class which provide an `unrank` function). So far, the benefits coming from such refinements do not seem to justify the overhead in the complexity of the category hierarchy.

Also, all of this is not really specific to combinatorial classes. We could imagine generalizing this to any kinds of collections of objects, and mimic the category hierarchy of, e.g. `Aldor`.

By convention, all the subcategories of `Cat::CombinatorialClass` have a name of the form `Cat::XxxClass`. Right now, we have two sub categories of `Cat::CombinatorialClass`:

- `Cat::decomposableClass`
- `Cat::integerListsLexClass`

Those two categories are purely technical; they respectively provide wrappers around the generic domains `combinat::decomposableObjects` and `combinat::integerListsLexTools`, and allow one to factor out some routine code. For example, `combinat::partitions`, `combinat::integerVectors`, and `combinat::compositions` are in the category `Cat::integerListsLexClass`, which takes care of the parsing of common options.

Further naming comments

The names of the domains `combinat::integerListsLexTools` and `combinat::decomposableObjects` are quite different. This reflects the fact that those two domains do not play the same role. `combinat::decomposableObjects` is a parametrized domain whose instantiations represent combinatorial classes, whereas `combinat::integerListsLexTools` essentially is a collection of tools with a scarce interface, geared toward speed and internal use.

The name `Cat::integerListsLex` is too general, since this category contains only the combinatorial classes described using length, bound, and slope constraints. For example, the elements of `combinat::permutations` are integer lists and are naturally ordered lexicographically; however this combinatorial class is *not* in `Cat::integerListsLex`. Badly enough, we have not found a better name that would not be too long. Unless someone comes up with a clever suggestion we will stick to this name.

We use `Next` for the name of the method that computes the next element in a combinatorial class. This is not coherent with `first`, `last` and with the general convention that a method name start by a lowercase letter. Badly enough, `next` is a reserved keyword, and we cannot use it as method name in MuPAD.

3.5 Representing combinatorial algebras

What is a combinatorial algebra after all?

Let us start by a precise definition for the term *combinatorial algebra* that we have used so far in a rather informal way.

Given a combinatorial class C, and a ring R, one can define the *free module F with basis indexed by C over the ring R*; an element of F is a formal finite linear combination of elements of C with coefficients in R. Alternatively, an element of F can be interpreted as a function from C to R with finite *support*; that is a function which is zero except on finitely many elements of the basis C.

For example, here is an element of the free module with basis indexed by partitions, over $\mathbb{Q}$:

$$x := 4[3,2,1] + 3[2,1,1] + 1/4[1,1,1,1]\,.$$

Polynomials are another typical example of free modules, and we extend the usual definitions used for polynomials. The *coefficient* of $[1,1,1,1]$ in x is $1/4$; we call the partition $[3, 2, 1]$, seen as an element of F, a *term*; $4[3, 2, 1]$ is a *monomial*; finally, the *support* of x is the set of the partitions with non-zero coefficients, that is $\{[3, 2, 1], [2, 1, 1], [1, 1, 1, 1]\}$.

By *combinatorial algebra* we mean such a free module, together with some extra algebraic operations (a product, coproduct, antipode) which makes it an algebra, a bialgebra, or a Hopf algebra. Those operations are typically defined by linearity on the basis.

With this definition, we have distinguished a special basis of the combinatorial algebra. Most of the time, a combinatorial algebra (like the algebra of symmetric functions) will actually have several interesting basis (Schur functions, power-sum functions, ...), all of them indexed by C. The underlying free module remains the same, but the operations will vary accordingly. Changing from one basis to the other is one of the fundamental operations.

Why use strong typing?

Traditionally in computer algebra systems (say with `SF`, `ACE` or μ-`EC`), symmetric functions have been represented by symbolic expressions:

```
>> p[2]*p[1];
   muEC::SYMF::Top(p[2]*p[1]);
   muEC::SYMF::Tos(p[2]*p[1])

                                 p[1] p[2]

                                  p[2, 1]

                             s[3] - s[1, 1, 1]
```

This is also the approach used for polynomials, and more generally for Ore-algebras in `Maple`. This has several advantages:

- This is simple, and requires (at first) very little programming and computer algebra knowledge from the user;
- The syntax for constructing elements is terse;
- All the standard tools for manipulating expressions such as `factor`, `expand`, `simplify`, ... are instantly available;
- One can easily manipulate factored expressions which mix different basis.

However, this approach has also serious drawbacks:

- The syntax for manipulating expanded expressions is lengthy; one cannot simply write (a*b);

- The tools for manipulating expressions do not know what they manipulate, and may do invalid operations; for example, symbolic expressions are usually considered as commutative, which may yield incorrect results when computing with non commutative symmetric functions. Staying on the safe side may require a fair amount of knowledge about the system from the user, which is not acceptable for beginners.
- Programming a function which deals with elements of the free module requires a fair amount of work just to parse the expressions on input (75% of the code in the symmetric functions package in ACE is related to this); one option to reduce the amount of code, is to first convert the input into an internal representation before manipulating it; however this means that the elements are converted back and forth all the time, which has a non-negligible computation cost.
- The data structure is not hidden, and there is no room left for optimization;
- There is a risk of conflict if two combinatorial algebras use the same name for their basis (e.g. all generalizations of the symmetric functions have some kind of elementary functions, which one would like to display as `e`). In particular, one cannot mix two algebras that use the same basis name in the same expression.
- One cannot easily choose the coefficient ring (think of symmetric functions with coefficients in a finite field, free symmetric functions with coefficients being themselves symmetric functions).
- One cannot easily hide and factor out the complexity, and let a user define his own algebra in a very short amount of code.

Altogether, this approach is fine when there are very few combinatorial algebras; however it does not scale to a dozen predefined algebras (that's our short-term goal) plus myriads of user-defined algebras. In practice, this is one of the main reasons why the `ACE` project stalled when defining many new algebras became a must.

It was time for a complete redesign and rewrite of the package. We will see that using strong typing allows for circumventing all those drawbacks, without loosing too much of the advantages. The choice of switching to MuPAD was largely influenced by the strong integration of their domains/axioms/category mechanism in their system, that allows for strong typing, and object-oriented techniques.

Representing free modules

The first thing to do is to choose the internal data structure to store an element of a free module. There are different possible implementations without a clear cut advantage of one over the others (as a parallel, in `C++` there exists two implementations of association tables: `map` using sorted lists and `hash_map` using hash tables). We provide several of them:

`Dom::FreeModuleTable(R, Basis)` An element `x` is stored as an association table (`DOM_TABLE`). For example, here is the internal representation of an element `x`:

```
>> F := Dom::FreeModuleTable(Dom::Rational, combinat::partitions):
   x := 4*F([3, 2, 1]) + 3*F([2, 1, 1]) + 1/4*F([1, 1, 1, 1]);
   extop(x)

      4 B([3, 2, 1]) + 3 B([2, 1, 1]) + 1/4 B([1, 1, 1, 1])

                      table(
                        [1, 1, 1, 1] = 1/4,
                        [2, 1, 1] = 3,
                        [3, 2, 1] = 4
                      )
```

Since any MuPAD object can be used as index of a table, there is no restriction on the basis elements. Accessing the coefficient of a term is constant time.

`Dom::FreeModulePoly(R, Basis)` The kernel polynomial objects of MuPAD (domain `DOM_POLY`) are stored in a sparse non-recursive way using sorted lists of monomials with a fixed number of variables. If one forgets about the product, this provides a sparse data structure which is both compact in memory and very fast for linear operations. Typically, the MuPAD sparse matrices (domain `Dom::SparseMatrix`) use univariate polynomials internally as internal representation for sparse column vectors.

Similarly, an element `x` of `Dom::FreeModulePoly(R, Basis)` is stored using a polynomial:

```
>> (F := Dom::FreeModulePoly(Dom::Rational, combinat::partitions);
   x := 4*F([3, 2, 1]) + 3*F([2, 1, 1]) + 1/4*F([1, 1, 1, 1]));
   extop(x)

      1/4 B([1, 1, 1, 1]) + 3 B([2, 1, 1]) + 4 B([3, 2, 1])

                              3        2
                 poly(1/4 _X  + 3 _X  + 4 _X, [_X])
```

To achieve this, one needs to be able to represent an element of the basis using an exponent vector (an integer, or a fixed-length list of integers). This is trivial when the basis readily consists of fixed length lists of integers (integer vectors, standard permutations of a given n, ...). Otherwise, the user may provide a pair of functions `rank` and `unrank` that does the conversions. By default, the system creates a dummy pair of such functions: the `rank` function associate in turn the numbers `1,2,3,...` to each new object it encounters. For example, the rank of `[3,2,1]`, `[2,1,1]`, and `[1,1,1,1]` above are respectively `1`,`2`, and `3`, that corresponds to the order in which the corresponding elements of `F` have been created.

This representation is very fast for linear operations. Furthermore, if univariate polynomials are used with the variable `_X` (this is the default), the data structure coincides exactly with the one used by `Dom::SparseMatrix`. This allows for zero-cost conversions to and from sparse vectors, for doing linear algebra.

Accessing the coefficient of a term in an element `x` is logarithmic in the number of terms of `x` (in the multivariate case, with MuPAD $< 3.0.0$ this is linear instead of logarithmic).

Ranking and unranking is only done for conversions, and computing products. In practice, the overhead with the dummy implementation seems to be negligible, and largely compensated by the fact that most operations deal with integers.

`Dom::FreeModuleList(R, Basis)` Thanks to Stefan Wehmeier, (Univ. Paderborn) and Werner M. Seiler (Univ. Karlsruhe), this was mostly already implemented in the MuPAD library under the name `Dom::FreeModule` since 1997. An element is represented by a sorted list of terms. For example, here is the representation of the element `x` above: `[[4, [3, 2, 1]], [3, [2, 1, 1]], [1/4, [1, 1, 1, 1]]]`.

Obviously, there needs to be an order on the basis elements (`Basis` should be a `Cat::OrderedSet`). Accessing a leading or trailing term is constant-time; accessing the coefficient of a given term in an element `x` is logarithmic in the number of terms of `x`.

All those implementations are in the category `Cat::ModuleWithBasis` which defines a unified interface. So, one can change the underlying implementation at any time. Unless you have a specific reason to choose one of the implementations, just use the default implementation `Dom::FreeModule(R, Basis)`.

Representing combinatorial algebras on a given basis

Once the underlying linear structure is implemented, it is very easy to construct functions on a free module by linearity / bilinearity / multilinearity on the basis (see the examples) using the utilities from `operators`. So, implementing operations like products, coproducts, antipodes usually boils down to implement the underlying combinatorics. Note that we do not have yet a general construction for the tensor product of two free modules `F1` and `F2`, but you can emulate one by hand by building a free module whose basis elements are pairs of basis elements of `F1` and `F2`.

Altogether, this allows to implement a domain `F` for the elements of a combinatorial algebra expanded on a given basis (e.g. the domain of symmetric functions expanded on the Schur functions, or the domain of symmetric functions expanded on the power-sum functions). The product of two elements of `F` is automatically expanded on the basis, and belongs again to `F`. Such a domain belongs to the category `Cat::AlgebraWithBasis` (later on, there will be categories such as `Cat::BialgebraWithBasis`, `Cat::HopfAlgebraWithBasis`).

Representing combinatorial algebras with several bases

A combinatorial algebra with several natural bases will be represented by several domains. Converting from one basis to another is an essential operation, and it is strongly desirable to be able to mix in the same expression elements that are expressed in different basis. This can now be achieved through overloading and the definition of appropriate implicit conversions (see the demonstration).

Developing the underlying tools that allow to do this seamlessly required a fair amount of work. Indeed, one parameter overloading in MuPAD works fine by delegating the work to the corresponding method of the parameter; on the other hand, the plain system essentially does not help much for multi-parameter overloading, which has to be carefully done by hand in each and every library (think about computing a product where the operands are in turn a symmetric function on the p basis, an integer, a symmetric function on the e basis, and a rational number). So we had to specify and implement a new mechanism that takes care of implicit conversions, and multi-parameters overloading. We essentially mimicked ideas taken from the GAP system [?], as well as from the static overloading mechanisms of standard languages like `C++`. The main difficulty was to choose the right level of generality, so as to make the system powerful enough, yet simple, safe, and sound. Our choices were largely influenced by our practical experience with the kind of computations we have in mind. This new overloading mechanism is being discussed for integration and systematic use in the MuPAD library. We are not at all specialists of this sensible subject, so comments and suggestions about this are very welcome. For details, see:

`http://mupad-combinat.sf.net/doc/html/operators/overloaded.html`

This mechanism is currently pretty rudimentary and limited. However, we have been playing with it intensively, and have done some really hairy stuff based on it. The semantic has proven so far to be safe, sound and robust, while really much more practical and powerful than the plain overloading mechanism. The time overhead is reasonable; if it became an issue, the specifications leave quite some room for optimisations, in particular by inclusion in the MuPAD kernel.

When defining combinatorial algebras with several bases, we organize the code in a fairly standardized way, which takes care of various technical issues such as initialization or parameterization of the algebra by the coefficient ring. We urge the interested reader to check out the code in the examples library, and

in particular:
`http://mupad-combinat.sf.net/lib/EXAMPLES/SymmetricFunctions.mu`

Conversions to and from expressions

Having strong safeguards is essential so that a beginner can run computations with confidence. However, one of our motto is that the system should not try to be too clever, and in particular should always leave a way for the user to take over the control (and the responsibilities!). Indeed, there always are situations where the user knows that a given operation, invalid in general, happens to be perfectly legal in the current context. Most of the time, this can be taken care of by systematically providing conversions to and from symbolic expressions that the user may manipulate at his convenience. However, there is no clearly-defined representation of elements of non-commutative algebras using symbolic expressions, because MuPAD (as most other systems) assumes that the latter are commutative. Just to give a flavor of the issue: which commutation rules should be applied automatically by the system in the expression `2*e[1]*q*3*f[3]`? Suggestions are very welcome here.

Compact notations

Throughout the tutorial, we have used fairly lengthy notations for constructing elements of combinatorial algebras. For example, to define the first symmetric power-sum, we wrote `S::p([1])`. This is fine in a tutorial when safety is at a premium, but in everyday's use, having terse notations is highly desirable. We are currently experimenting several tricks that allow for simultaneously using `p` to represent the domain of symmetric functions in the `p` basis, and `p[1]` for creating the first symmetric power-sum, while still being able to convert symmetric functions into symbolic expressions containing literals such as `p[1]`. As soon as we will have more experience with this, we will describe the recommended practice in the Tips and Tricks section of the reference manual.

Acknowledgments

We would like to thank all the authors who directly or indirectly contributed ideas and code to `MuPAD-Combinat`: Christophe Carré, Sébastien Cellier, Sylvie Corteel, Alain Denise, François Descouens, Teresa Gomez-Diaz, Isabelle Dutour, Houda Abbad, Daniel Krob, Axel Kohnert, Alain Lascoux, Éric Laugerotte, Conrado Martinez, Xavier Molinero, Jean-Christophe Novelli, Vincent Prosper, Frédéric Sarron, Jean-Yves Thibon, Chan Ung, Sébastien Veigneau, and Paul Zimmerman. We are very grateful to the members of the MuPAD team for their continuous support, and in particular Christopher Creuztig, Klaus Drescher, Ralf Hillebrand, Walter Oevel, and Stefan Wehmeier. We also would like to thank Alessandra Frabetti and Sergei Lando for their help in proofreading this paper. `MuPAD-Combinat`is hosted on the `sourceforge.net` open-source support center, and this has proved invaluable; thanks to them.

Last, but not least, we would like to express our special gratitude to Volker Strehl for his kind and strong support and clear sighted suggestions for putting this material into a form suitable for publication.

References