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

F. Hivert, N. Thiéry

18th March 2003

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 (`http://mupad-combinat.sf.net`). 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 part of the `MuPAD-Combinat` reference manual `http://mupad-combinat.sf.net/doc/html/`. We refer to this document for details and references on the algorithms used.

Contents

1 Introduction

MuPAD-Combinat is an open-source algebraic combinatorics package for the computer algebra system MuPAD 2.0 and higher. The development started in spring 2001, and the package currently contains functions to deal with usual combinatorial classes (partitions, tableaux, decomposable classes, ...), Schubert polynomials, and characters of the symmetric group. 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. The core of the package is integrated in the official library of MuPAD since version 2.5.0.

Here are a few facts: about 40000 lines of code in high level language, 15000 lines of C++ (+80000 lines of C imported from Symmetrica), 350 pages of documentation (including 150 pages from mu-EC [Pro99]). Over the last year: 500 messages sent on the mailing-list, 150 of which coming from the MuPAD developers, 2000 visits of the web page and 200 downloads. The developing team includes 3 core developers (T. Gomez-Diaz, ingénieur de recherche, IGM, joined us recently), and 3 contributers (J.-C. Novelli (CNRS, IGM), Éric Laugerote and Christophe Carré (maîtres de conférence, Rouen), not counting the many authors of the software that were integrated.

Most of the code from F. Hivert and N. M. Thiéry derives from successful computer exploration while doing research on the following topics:

- Invariant theory of permutation groups [Thi99, Thi00, Thi01],
- Steenrod algebras and symmetric functions [HT02],
- Binary trees algebras such as the Loday-Ronco algebra, and renormalization in Quantum Electrodynamics [HNJY02],
- Representation theory and quantum groups, homological computations,
- Peaks Algebra and Hecke Clifford Algebras.

The package is organized as follow:

- The `combinat` library provides combinatorial functions, as well as tools for counting, generating, and manipulating classical combinatorial classes (partitions, compositions, tableaux, trees, ...). This includes in particular the former `CS` library, under the name `combinat::decomposableObjects`.
- The `IPC` library provides some tools for Inter-Process Communications.
- The `operators` library provides some tools for defining overloaded functions as well as extending functions by linearity, associativity, ...

- Several domains are provided in `Dom` for algebraic structures such as permutation groups, boolean and tropical algebras, or weighted automatons.
- Several examples of use, such as some predefined combinatorial algebras (symmetric functions, free quasi symmetric functions, the Loday-Ronco algebra), are provided in `examples`. You can also browse through the `experimental` stuff but this is at your own risk.
- Additionally, the following libraries are imported verbatim from the former muEC package:
  1. `muEC::PART` - library for partitions
  2. `muEC::COMP` - library for compositions
  3. `muEC::TYP` - library for type-checking of mu-EC objects
  4. `muEC::SG` - library for the symmetric group
  5. `muEC::SYMF` - library for symmetric functions
  6. `muEC::SFA` - library for symmetric functions on different alphabets
  7. `muEC::SP` - library for simple and double Schubert polynomials

  The package mu-EC is provided as-is, to ensure backward compatibility for former users of mu-EC. Its functionnalities are not yet fully integrated. For example, `muEC::PART::ListPart` is redundant with `combinat::partitions::list`; in the long term all the functionalities of `muEC::PART` which are missing in `combinat::partitions` should be moved into the later which has a better and more systematic design.

2 Motivations, history and status

The main purpose of this package is to provide an extensible toolbox for computer exploration for research in algebraic combinatorics. Let us start by explaining a little what we mean by algebraic combinatorics.

2.1 Combinatorics and Algebra

"Laplace discovered the remarkable correspondence between set theoretic operations and operations on formal power series, and put it to great use to solve a variety of combinatorial problems". – Gian-Carlo Rota [Rot75]

The intervention of algebra in combinatorics came very early. For example, the basic tools of formal power series allow to compute easily with the complicated recurrence relations that appear when counting combinatorial objects. In the same spirit, a classical approach to count objects with symmetries is to break those symmetries by putting labels on the parts of the object; then there is a group acting on the labels, and the problem is reduced to compute the number of orbits under the action of this group. This is the classical method of Pólya enumeration, which is treated extensively in Kerber's book "Algebraic combinatorics via finite group actions" [Ker91].

Conversely, in modern algebra, the classification problems in group theory or in representation theory, have given risen to many constructions (should we say algorithms ?) of algebraic objects from combinatorial inputs such as partitions, Young diagrams, Dynkin diagrams, finite oriented graphs (also known as *quivers*).

In both cases, there is a strong need to compute with algebraic objects, say vectors spaces or algebras, whose basis are indexed by combinatorial objects, which we will call *combinatorial algebras*.

One of the first and most fundamental object in algebraic combinatorics is the algebra of symmetric functions. This was first understood by Mac-Mahon which is perhaps the father of the modern algebraic combinatoric. The interest of the algebra of symmetric functions comes from its extremely rich algebraic structure, starting from the fact that it's not only an algebra but also a Hopf algebra – a *Hopf algebra* is an algebra whose linear dual is also an algebra, with some compatibility between these two structures; essentially, there is a "decomposition law". The importance of Hopf algebra in combinatorics was perhaps first stressed by G.-C. Rota who pointed out how many combinatorial objects naturally posses a Hopf structure, because they can be composed and decomposed. In fact, under this combinatorial Hopf structure, many things that we would like to count transform naturally, yielding new tools in combinatorics. The study of symmetric functions was the starting point of a domain of combinatorics which is devoted to the study of combinatorial Hopf algebras. This is a rapidly growing field, and it has application in many different areas, such as representation theory in mathematics with quantum groups, and renormalization in quantum field theory in physics.

2.2 A need for a toolbox for computer exploration

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 the very least they can be sent to the *Online Encyclopedia of Integer Sequences* [] for comparison with other well known sequences. In general, using a computer allow for studying large scale examples (in combinatorics, the size of the examples usually grow very quickly!). This can help suggest conjectures, check them for likeliness, or find counter-examples.

Here are, in our experience, the specificities of the needs to run such computations:

- Our ultimate goal is to do research, not to write software. However, to do this research we need appropriate tools. Hopefully, in the middle term, sharing those tools and the associated development time with others is a way for us to save time for more research.

- The development cycle ought to be short; to stress on this, here is the schedule of one of our (ideal) research days:

  10:00 Conjecture A is on the board;

  10:10 We have a first idea on how to test it on a computer

  10:30 The code is written; the laptop starts computing

  10:31 The first results are there; oops, a counter-example; the conjecture is indeed incorrect as is;

  10:45 A patched conjecture A' is formulated;

  10:46 The laptop starts running the adapted code;

  10:50 The first results are there; ok, maybe this conjecture is not so stupid after all; let's try to prove it;

  11:30 More results are there; but memory is running low on my laptop. Hey, see ? all the coefficients are integers. This gives a good hint for the proof. But this also suggests a much better algorithm for running the computation.

  12:30 The new code is written. Good, it runs 10 times faster, and uses far less memory. Let's start it on the server, and go for lunch.

  13:00 Back from lunch; many more examples have been checked in the meantime.

  15:00 Conjecture A' is proved; problem B(A') is on the board. the server is computing some examples overnight.

  17:00 The code written during the day is analyzed; the bits that can be reused are cleaned up and integrated into `MuPAD-Combinat` together with documentation and test files; everything is instantly available on the anonymous CVS server on `mupad-combinat.sf.net`.

- Most of the required combinatorial utilities are quite common (say, list all the partitions of a given integer, ...); however, there are so many potential common utilities that a combinatorial package will never be able to provide all of them, or at least not in an optimized form. On the other hand, such a package should make it easy to integrate new utilities by providing a well designed framework. Ideally, the package should act as a repository where users would include their utilities while they develop them for their own needs. Furthermore, the package should provide versatile tools that makes it easy to define new combinatorial classes, new algebras, and so on.

- 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. Of course, ideally the code should be written in such a way to leave room for such specializations and optimizations. All of this speaks for a high level language which allows for writing code that sticks as much as possible to the mathematical way of thinking. Of course, this does not preclude the use of some modules written in a low level language like C for the critical sections, when there is a clear need for it.

- One often need general computer algebra tools, like linear algebra, integration, Gröbner basis, and so on. Interfacing with such tools should be as seamless as possible.

- The package should allow for different levels of usage:

  – 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, programing of complete libraries for new combinatorial classes and algebras;

  – Core hacking, implementation of generic algorithm, writing of external modules (say in C) for optimized speed, ...

- Being integrated in a well known and widely available system helps so that the user can work in his usual environment; this is particularly important for the first two levels of usage above.

Roughly speaking, there are currently three kinds of tools available:

- Tools for enumerative combinatorics like the `Combstruct` [Zim94] package for `Maple` [CGG+88], the `CS` [DDZ98] package for MuPAD, or `Calico`. The combinatorial objects are described by specifications (like context free-grammars), and the goal is to count, generate, or draw them at random. Another example is the `MOLGEN`program from Kerber's team to deal with chemical structures.

- Tools for analyzing generating functions like the `gfun` and `Mgfun` `Maple` [CGG+88]packages by B. Salvy, P. Zimmermann, E. Murray and F. Chyzak, the `RATE` Mathematica library by Christian Krattenthaler.

- Tools for computing in various combinatorial algebra, especially the algebra of symmetric functions. The most famous one is Stembridge's `SF`package for `Maple` [CGG+88]. A more ambitious package for `Maple` [CGG+88], called `ACE` [Vei98], has been developed in Marne-la-Vallée, under the lead of S. Veigneau, and has been partially ported to MuPAD under the name `mu-EC` [Pro99]by Vincent Prosper. Weybourne also wrote an interactive program called `Schur`for calculating properties of Lie groups and symmetric functions, with a view toward physics. A. Kohnert, wrote `Symmetrica` [KKL], 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. Finally, one should mention the `Maple` [CGG+88]library for Gröbner basis computations in ORE-algebras by F.Chyzak.

See `http://www.mat.univie.ac.at/~slc/divers/software.html` for a complete list of related software.

2.3 The birth of `MuPAD-Combinat`

An itch to scratch

We (F. Hivert and N. M. Thiéry) had both written relatively large pieces of code for our own computations and discovered, *a posteriori*, that the core tools of our libraries were essentially the same. Furthermore, several excellent packages we used, like `ACE` [Vei98], `mu-EC` [Pro99]or `CS` [DDZ98], were slowly dying out by lack of proper maintenance and distribution.

So, we decided to put our work in common and, in order to avoid as much as possible the *Yet Another Combinatorial Package (tm)* effect, we tried to analyse somewhat the situation. Basically, here are some of the problems we face:

- There are too many potential platforms for combinatorics (Maple, Mathematica, MuPAD, GAP, Magma, Axiom, Aldor, ...), with little tools for exchanges between them (so far, will Open Math save us one day ?). All those platforms have advantages, and all of them have drawbacks: not open source, expensive, not widely available, too specialized, *very* bad programming language, ...

- Existing packages for algebraic combinatorics are designed to compute efficiently with specific algebras. However they are not easily extensible because they do not provide tools to construct new combinatorial algebras.

- There is a lack of communication and software exchange in the community: too many things get *reimplemented* over and over. Many little pieces of code just stay in users home directories without being *integrated*, *distributed* and *reused*.

- The long term *software maintenance* is sloppy: often a package is developed by a student, which then leaves for better skies. Shortly the package gets incompatible with the next version of the platform, and finishes in the trash bin.

An opportunity

Some time ago, N. M. Thiéry had made contacts with the MuPAD group, in order to integrate his library `PerMuVAR`, or parts of it, to the official release of MuPAD. This library contained several combinatorial routines, and he suggested some reorganization of the (rather minimal) combinat package of MuPAD, in order to ensure a better integration. His offer was warmly welcome, and the MuPAD group actually decided to make him the official maintainer of the `combinat library`, with full freedom for reorganizing and expanding it.

Is MuPAD the best platform for our needs ? Technically, it has proven so far to be reasonably adapted to our requirements. The language is quite clean, and seems to cope properly with change of scales. Most important, the collaboration with the MuPAD team is working smoothly; in particular they have been open-minded with respect to our (numerous) suggestions of improvements, including some sensitive kernel changes to polish the semantic of domains.

On the other hand, MuPAD is not open-source. We are not happy with this situation, and actively negotiating with them about this; for example, the recent release of the MuPAD library under an open source license was a precondition we had set for the inclusion of `MuPAD-Combinat`in this library. Our impression is that they are, in theory, very favorable to open-source ideas but in practice, they judge that they could not currently open the kernel without putting their very survival at great risks.

Development model and plan

Our bet is that what is needed above all is a *good development model*. We want to see how well the open-source development model can apply to our community. The choice of open-source itself is obvious; however it's not clear whether the principles that made the success of general interest open-source software will apply because the targeted audience is relatively small. The main challenge is to reach quickly the critical mass above which the package will attract users and developers, and will grow on its own by the users contributions. To this end,

- Can we integrate other developers ?
- Can we integrate other packages ?
- Can we interface with other software ?

The point is *not* to take the control of other packages. It's just to drop a little bit of glue between packages to help them cooperate together.

Anyway, the plan is to quickly extend the features of `MuPAD-Combinat`, while keeping it clean, organized, stable, and well documented. To this end, the core developers concentrate on the following:

- Provide a sane framework, where external code can easily be integrated;
- Maintain the package, make it easy to install, ...
- Coordinate the contributions, and provide support to those who wish to contribute.

The real coding work ought to be done by users, so *we need your help*! See the web page `http://mupad-combinat.sf.net` for how you can contribute.

Note that there are many features that are not yet completely documented, or that would be trivial to add in a very short amount of time. So, if you miss a specific feature to get started, please feel free to ask for it. We are ready to invest some time in helping out new users: hopefully they will become future contributers. And in general, we really are looking for feedback.

2.4 Current features

So basically, what do we provide ? The first part is a classical combinatorial package to count, enumerate, or manipulate standard objects (partitions, ...). Note that, in order to build some new combinatorial classes, we provide for the moment three computational engines, one for dealing with integer vectors with prescribed constraints (monomials, compositions, partitions, Dyck paths...), one for generating linear extensions of posets (standard young tableaux, standard ribbons, ...), as well the former `CS` library by S. Corteel, A. Denise, I. Dutour, and P. Zimmermann which allows to enumerate classes of objects described by a context-free grammar. Several specific packages actually makes use of these engines.

The 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 parsing algebraic elements, extensions of functions by linearity, bi-linearity or associativity, conversions between different bases... Thus you can concentrate on the combinatorics rather than on the programming!

Finally, as example of use and application, we provide a library for the algebra of symmetric functions, and we have (currently undocumented) implementations for 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 mid-term (or short term if you help us!) we plan to provide predefined libraries for the algebra of non commutative symmetric functions, the free symmetric algebra, the algebra of quasi-symmetric functions, 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, ...

3 A guided tour through MuPAD-Combinat

3.1 Two examples of combinatorial algebras

The main purpose of this package is to provide an extensible toolbox for computer exploration for research in algebraic combinatorics. The main objects of interest are combinatorial (Hopf) algebras. In the sequel, we present two examples of such combinatorial algebra.

We load the MuPAD-Combinat package, and define a shortcut for the algebra symmetric functions:

```
>> package("Combinat", Quiet):
   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
```

Computing the product of two such 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]
```

Now, we are working in an algebra whose basis is indexed by partitions, and we want to compute efficiently in this algebra.

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's implemented as a combinatorial algebra having binary trees as basis.

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

For example, take the two following trees:

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

                              o
                             / \
                                \
```

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

```
   o
 /  \
/ \
```

You can make a formal linear combination of them:

>> t3 := 2*t2 + 3/4*t1

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

or take their product:

>> t2*t1

```
   o           o              o        o              o
 /  \        /    \        /   \     /    \        /   \
/ \   \   + / \ /  \  +  /  \   \ + / \  / \  +  /   \  \ +
     / \         \  \   / \  \          /   \   / \ /
        \

         o
        / \
      /    \
    /  \
   / \
```

Here is a more complicated product in this algebra:

>> t3*t2

```
     o               o               o
    / \             / \             /      \
       \                   \          /     \
3/4        \  + 3/4    /    \ + 3/4    \      +
         /  \            \               \
        / \             / \             / \

              o         o               o
        /      \       / \             /      \
       / \                    \          /     \
   3/4    \      + 3/4      /  \ + 3/4      \    +
```

```
         \             /  \            /  \
        / \             \               \

           o         o                     o
     /      \       /      \         /      \
    / \                  /  \       /    \
3/4      \    + 3/4   /   \   + 3/4   /   \   +
       /  \            \               \
        \               \               \

           o         o               o
         /  \       / \             /      \
     /    \                \          /     \
3/4 / \       + 3/4      /  \ + 3/4      \    +
       \                / \             / \
        \              /               /

           o         o                     o
     /      \       /      \         /      \
    / \                  /  \       /    \
3/4      \    + 3/4   /   \   + 3/4   /   \   +
        / \             \               \
       /               /               /

           o         o                     o
         /  \       /      \         /      \
     /    \              /  \       /    \
3/4 / \       + 3/4     / \   + 3/4     / \   +
        \             /               /
       /               \               \

           o               o
         /  \            /  \        o
     /    \             / \        /  \
3/4 /   \     + 3/4  /        + 2 / \     \  +
      /             / \                 /  \
       \               \               / \

     o                   o         o                   o
   /      \         /     \      /      \         /     \
2 / \ /    \ + 2  /  \      + 2 / \   /  \ + 2  /    \    +
        \        / \   \            /  \       / \ /  \
       / \            / \            \              \
```

```
          o         o                   o              o
        /  \      /      \         /     \           /  \
2    /   \   + 2 / \   /  \ + 2  /    \    + 2    /   \   +
   /  \               / \       / \  / \        /   \
  / \  \             /              /          / \ /

          o
        /  \
       / \
2    /
   /  \
  / \
```

3.2 MuPAD-Combinat, step by step

We now describe in more details how all of this works. In the following, we assume that the package `MuPAD-Combinat` has been loaded into MuPAD and, for shortening the notations, that the library combinat has been exported. We also assume that the reader is somewhat familiar with the MuPAD syntax, and refer to the MuPAD tutorial for details.

```
>>  package("Combinat", Quiet):
    export(combinat):
```

Using predefined combinatorial functions and classes

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, like partitions. For each such class, there is a sub library of `combinat`. We can use the library `combinat::partitions` to list all the 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's draw the partition `[3,2]` using boxes (French/cartesian notation):

>> partitions::prettyPrint([3, 2])

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

Now, 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::prettyPrint)

```
-- +---+---+      +---+---+      +---+---+      +---+---+
|  | 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 a lot of 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's look in more detail at the library `combinat::partitions`. We can count partitions:

>> partitions::count(i) **$** i = 1**..**10

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

list them with some extra conditions:

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

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

or compare them:

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

```
FALSE
```

An important feature of MuPAD-Combinat are the so-called *generators*, which allow to run through huge lists of combinatorial objects without expanding the full lists into memory:

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

```
[4]

[3, 1]

[2, 2]

[2, 1, 1]

[1, 1, 1, 1]

FAIL
```

This becomes 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 libraries 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])

      [[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]]
```

They actually use the same computational engine `combinat::integerListsLexTools` in behind the scene:

```
>> partitions::list(9, MinPart = 2, MaxPart = 5);
   integerListsLexTools::list(9, 0, infinity, 2, 5, -infinity, 0)

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

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

The same computational engine 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 for labelled binary trees, and it could be reused as well for generating standard skew tableaux, standard ribbons, and so on (any volunteers?).

We also incorporated and extended the former `CS` library by S. Corteel, A. Denise, I. Dutour, and P. Zimmermann. This library allows for manipulating any 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:

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

```
>> 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)]
```

Uh, this is not very readable. Let's improve this 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 rewriting rules in the grammar:

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

```
[[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]]
```

This seems to work nicely. Let's 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 Fibonnaci 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
```

Defining new combinatorial classes

Let's define one of the most trivial combinatorial classes:

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

     info := "Domain 'nonNegativeIntegers: the class of non negative integers";

     // The domain of the elements of this class
     domtype := DOM_INT;
     // The type of the elements of this class
     type := Type::NonNegInt;

     // The size of an integer is itself
     size := n -> n;
     // There is exactly one integer of size n
     count := n -> 1;
     list := n -> [n];
     // No need to define generator; it's defined via list by default
   end_domain:
   testtype(x, nonNegativeIntegers), testtype(-3, nonNegativeIntegers), testtype(3, nonNegativeIntegers);
   nonNegativeIntegers::count(4);
   nonNegativeIntegers::list(4)
```

```
FALSE, FALSE, TRUE

1

[4]
```

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.

Exercise: Improve the definition above so that `nonNegativeIntegers::count(-1)` raises an error instead of returning -1.

It's often practical to define a subclass of an existing class. Here we show how to define the class of the standard permutations of `[1,2,3]`:

```
>> domain permutationsOf123
       // Inherits all the methods from combinat::permutations
       inherits combinat::permutations;
       // This is a combinatorial class
       category Cat::CombinatorialClass;
       // This is a facade domain
       axiom Ax::systemRep;

       info := "Domain 'permutationsOf123': the class of the permutations of [1,2,3]";

       // Redefinition of type
       type := Type::Intersection(combinat::permutations::typeStandard,
                 Type::Predicate(l -> bool(nops(l)=3)));
       // Redefinition of count
       count := () -> combinat::permutations::count(3);
       // Redefinition of generator
       generator := () -> combinat::permutations::generator(3);
       // No need to redefine list, since it's defined via generator
   end_domain:
```

Let's use this new combinatorial class:

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

                       FALSE, FALSE, FALSE, TRUE

                                   6

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

    [3, 2, 1]]
```

Exercises:

- Define a parametrized domain `permutationsOf(l)` for the permutations of the list `l` (or of `[1..l]` if `l` is an integer);
- Define the combinatorial class of prime integers;

- Define a parametrized domain `classOfSet({a,b,c})` that creates the trivial combinatorial class composed of the elements `a`, `b`, and `c`. You may assume that all the elements of the set are identifiers, and define the `domtype` entry as `DOM_IDENT`.

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
      inherits Dom::BaseDomain;
      // The objects of this class are defined by a grammar
      category Cat::DecomposableClass;
      // This is a facade domain
      axiom Ax::systemRep;

      info := "Domain␣'FibonacciWords':␣the␣class␣of␣Fibonacci␣words";

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

      // The type of the elements of this class
      type := Type::Predicate(
   proc(w) // a procedure that test if w is a Fibonacci word
      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;
   TRUE;
            end_for;
   end_proc);

      // The function that returns the size of the elements of this class
      size := nops;

      // The grammar which defines the objects of this class
      grammar := decomposableObjects(
   [FiWords = Alias(FiWordsRec, DOM_LIST),
        FiWordsRec = Union(Epsilon(),
                           Atom(B),
                           Alias(Prod(FiWordsRec, Atom(A)), op),
                           Alias(Prod(FiWordsRec, Atom(A), Atom(B)), 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);
   FibonacciWords::count(4);
   FibonacciWords::unrank(2, 4)

 [[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]]

                              8

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

Exercises:

- Add two methods `toString` and `fromString` which converts a Fibonacci word to and from a string.
- Define a parametrized domain `FibonacciWords(A,B)` for the Fibonacci words on the letter `A` and `B`.

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 bases. For example, it's the free commutative algebra on the symmetric powersums:

```
>> p1 := S::p([1]);
   p2 := S::p([2]);
   p3 := S::p([5,4]);
   p2*p1*p2*p3

                                  p[1]

                                  p[2]

                                p[5, 4]

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

Note that any expression is immediately expanded by the system:

```
>> (p3 + 3*p2) * (1/4*p1 - p2)

     1/4 p[5, 4, 1] - 3 p[2, 2] - p[5, 4, 2] + 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);
   S::p

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

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

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

```
>> f := (expr(p3) + 3*expr(p2))*(1/4*expr(p1) - expr(p2))

                              / p[1]        \
            (3 p[2] + p[5, 4]) | ---- - p[2] |
                              \   4         /
```

Here is how to convert the expression back into an object of the domain `S::p`:

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

     1/4 p[5, 4, 1] - 3 p[2, 2] - p[5, 4, 2] + 3/4 p[2, 1]

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

This requires the use of a small trick because of the indexed notation for the basis elements: 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]).

A symmetric function `f` can be expanded on any alphabet (e.g. set of variables) by simple evaluation:

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

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

Of course, `S` provides the other classical bases of the 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. Since there will be now a parameter `t`, we take the 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
```

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 algebra over the rational numbers. Its basis is indexed by words, and the product of two basis elements is obtained by concatenating the corresponding words:

```
>> domain FreeAlgebra // line 1
       inherits Dom::FreeModule(Dom::Rational, combinat::words); // line 2
       category Cat::AlgebraWithBasis(Dom::Rational); // line 3

       initDomainAdd := dom::initAlgebraWithBasis; // line 4
       one := dom::term([]); // line 5
       mult2Basis := dom::term @ _concat; // line 6
   end_domain: // line 7
```

We will explain the bits of this definition in a minute after a few examples of usage. Let's 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]) + 9 B([]) + 2 B([a, b, c, a, b, c]) +

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

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. Line 1 states that we are declaring a new *domain* called FreeAlgebra (a new class in usual object oriented terms). Line 2 let the domain FreeAlgebra inherits its implementation from the free module over the rationals (Dom::Rational) with basis indexed by words (combinat::words). Line 3 states that FreeAlgebra is actually an algebra with a distinguished basis; this allow, in particular, to define by default the multiplication by linearity on the basis. Line 4 is just some required black magic; you may safely ignore it. dom refers to the domain being defined. dom::term is a constructor that takes an element of the basis, and returns it as an element of the domain. So, line 5 defines that the empty word is the unit of `FreeAlgebra`. Finally, line 6 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 all there is to it.

Let's define the free commutative algebra:

```
>> domain FreeCommutativeAlgebra
       inherits FreeAlgebra;
       category Cat::AlgebraWithBasis(Dom::Rational);

       straightenBasis := dom::term @ sort;
       mult2Basis := dom::straightenBasis @ _concat;
   end_domain:
```

As you can see, we simply inherit from `FreeAlgebra` and just redefine the product: using domains (and in general object oriented programming) helps reusing code. 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 the order. A priori `B([a,b])` and `B([b,a])` represent the same element of the algebra.

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, we could have used `listlib::merge` instead.

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

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

```
B([a])

B([a])
```

and even have the exact same internal representation:

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

```
TRUE
```

However, they are not equal, because they are not of 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, to avoid trouble if 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's apply this morphism to the sum of two words which only differ by a permutation:

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

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

                          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 `operators::overloaded::declareConversion`:

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

```
                          2 S([a, b, c])
 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 mechanisms we use for defining automatic conversions and overloaded operators and functions. Note that it's 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.

To continue our exploration, we define variants of 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 indexed respectively by compositions and by partitions. Technically, we don't let the second domain inherit any more from the first one, as they don't share the same basis.

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

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

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

The reader will have recognized respectively the commutative and non commutative symmetric functions, expanded (for example) on the elementary symmetric functions, hence the basis names. To shorten the notations, we define two aliases, and declare the same 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]);
   y := SF ([1, 3, 2])
```

```
E[1, 3, 2]

e[3, 2, 1]
```

Let's analyze the difference with our previous implementations of the free algebras. First, we opted for 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`: `exprTerm` is the method of the domain which 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 the integers in the partitions are sorted decreasingly. Here, this is 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 be conveniently implemented using the `zip` MuPAD function.

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

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

Let's 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 data representation changes. So, we implement the reciprocal conversions isomorphism 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's 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 mechanisms chose 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's enough to implement $2n$ conversions to be able to get all the $n(n-1)$ possible conversions. Of course, it's still possible to implement some extra direct conversions for better efficiency; when there are several ways to convert an element from one domain to another, the system always uses one of the most direct ones.

We conclude by demonstrating how to do some basic linear algebra, say to find the minimal polynomial of an element of the group algebra of the symmetric group of order 4:

```
>> domain AlgSymGroup4
        inherits Dom::FreeModule(Dom::ExpressionField(), combinat::permutations);
        category Cat::AlgebraWithBasis(Dom::ExpressionField());

        initDomainAdd := dom::initAlgebraWithBasis;
        basisName := hold(p);
        exprTerm := dom::exprTermIndex;
        one := dom::term([1,2,3,4]);
        mult2Basis := dom::term @ combinat::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;
    solve([coeff(y)], [a1,a2,a3,a4]);
    subs(z^0 + a1*z^1 + a2*z^2 + a3*z^3 + a4*z^4, op(last(1),1))
```

```
a3 p[4, 1, 2, 3] + a2 p[3, 4, 1, 2] + (a4 + 1) p[1, 2, 3, 4] +

   a1 p[2, 3, 4, 1]

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

                                    4
                               1 - z
```

4 The design of the MuPAD-Combinat package

4.1 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 >= 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 math 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 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.

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 returns a similar result but under different forms; the function that is the most usefull/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 modules methods (mult/multBasis/mult2/mult2Basis, straighten/straightenBasis, print/printTerm/printMonomial/printBasis).

4.2 Representing combinatorial objects and classes in MuPAD

A *combinatorial object* is, err, well, a combinatorial object; it's best described by examples: a partition, a tree, a tableau, a Dyck word, a composition 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 defines 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's 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 in MuPAD

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]);
   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)
```

```
                          DOM_LIST

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

In the MuPAD jargon, 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::binaryTrees([1, [1], [1]]);
   domtype(t);
   testtype(t, combinat::binaryTrees);
```

```
                              o
                             / \

                    combinat::binaryTrees

                            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 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 bug 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 in MuPAD

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's 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, we it's 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 for using 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 classes 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 which 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`

The reason of being of both those categories is purely technical; they respectively provide wrappers around the generic domains `combinat::decomposableObjects` and `combinat::integerListsLexTools`, and allow for factoring out some routine code. For example, `combinat::partitions`, `combinat::integerVectors`, and `combinat::compositions` are in the category `Cat::integerListsLexClass`, which entirely takes care of all the common option parsing.

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, while `combinat::integerListsLexTools` is more of a collection of tools with a scarce interface which is geared toward speed and internal use.

The name `integerListsLex` is too general, since it does not refer to all kinds of combinatorial classes composed of integer lists that can be ordered lexicographically, but those that can be described using length, bounds and slope constraints. For example, `combinat::permutations` is *not* in the category `Cat::integerListsLexClass`. However, we haven't 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.

4.3 Representing combinatorial algebras in MuPAD

What is a combinatorial algebra after all?

We use the term *combinatorial algebra* in rather informal way. To be more precise, given a combinatorial class C (say the partitions), and a ring (or field) R (say the rational numbers), one can define the *free module F with basis C over the ring R*; an element of F is a formal finite linear combination of elements of C with coefficients in R, like

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

. Alternatively, an element of F can be interpreted as a functionq from C to R with finite *support*; that is a function which is zero except on finitely many elements of the basis C.

Polynomials are a typical example of such 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 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, antipod) which makes it an algebra, bigebra or 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 (the schur functions, the powersums 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` [Vei98] or `mu-EC` [Pro99]), 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 ORE-algebras in `Maple` [CGG+88]. 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 (factor, expand, simplify, ...) are instantly available;
- One can easily manipulate factored expressions which mix different basis.

However this approach as also serious drawbacks:

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

- The tools for manipulating expressions don't 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.
- Programing 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 the generalizations of the symmetric functions have some kind of Schur functions, which one would like to display as `s`). In particular, one cannot mix two algebras which use the same basis name in the same expression.
- One cannot choose easily the coefficient ring (think of symmetric functions with coefficients in a finite field, free symmetric functions with coefficients which are 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` [Vei98] 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, which 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 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::FreeModule_Table(R, Basis)` An element `x` is stored as an association table (`DOM_TABLE`). For example, here is the internal representation of the element `x` above:

```
>> F := Dom::FreeModule_Table(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]) + 4 B([3, 2, 1]) + 3 B([2, 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::FreeModule_Poly(R, Basis)` The kernel polynomials 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 library (`Dom::SparseMatrix`) uses univariate polynomials internally for storing sparse vectors.

Similar wise, an element `x` of `Dom::FreeModule_Poly(R, Basis)` is stored using a polynomial:

```
>> (F := Dom::FreeModule_Poly(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 do this, one need 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 sees. For example, the rank of `[3,2,1]`, `[2,1,1]`, and `[1,1,1,1]` above are respectively `1`,`2`, and `3`, which 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`, which 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 and such. 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::FreeModule_List(R, Basis)` Thanks to Stefan Wehmeier, (Uni Paderborn) and Werner M. Seiler (Uni Karlsruhe) this is 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`.

NOTE: Note that those names are subject to changes; unless you have a specific reason to choose one of the implementations, just use the default implementation `Dom::FreeModule(R, Basis)`.

All those implementations are in the category `Cat::FreeModule` which defines a unified interface. So, one can change the underlying implementation at any time.

Representing combinatorial algebras on a given basis

Once the underlying linear structure is implemented, it's 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, antipods usually boils down to implement the underlying combinatorics. Note that we don't 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 allow 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 powersum 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::BigebraWithBasis`, `Cat::HopfAlgebraWithBasis`).

Representing combinatorial algebras with several basis

A combinatorial algebra with several natural basis will be represented by as many domains. Converting from one basis to another is an essential operation, and it's 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 allows to do this seamlessly was 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 taken care 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 [GAP99], 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.sourceforge.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 kernel. One issue that remains is where in the code the overloading of the operations should be declared (in the domain, in the category, ...). This is the role of the magic line `initDomainAdd := initAlgebraWithBasis;` in the definitions of the domains in Cat::AlgebraWithBasis; ideally this line should disappear in the future.

When defining combinatorial algebras with several basis, 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.sourceforge.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's 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 are always situations where the user knows that a given operation, which is 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 way yet for how to represent elements of non-commutative algebras using symbolic expressions; indeed MuPAD (as most other systems) assumes that the later are commutative. Just to give a flavor of the issue, which commutation rules should be applied by the system in the expression `2 * e[1] * q * 3 * f[3]` ? Suggestions are very welcome here.

Compact notations

Allover the tutorial, we have used fairly lengthy notations for constructing elements of combinatorial algebras. For example, to define the first symmetric powersum 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 allows for simultaneously using `p` to represent the domain of symmetric functions in the `p` basis, and `p[1]` for creating the first symmetric powersum, 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, Isabelle Dutour, Teresa Gomez-Diaz, Daniel Krob, Axel Kohnert, Alain Lascoux, Éric Laugerotte, Jean-Christophe Novelli, Vincent Prosper, Frédéric Sarron, Jean-Yves Thibon, Chan Ung, Sébastien Veigneau, and Paul Zimmerman. We also would like to thank the members of the MuPAD team for their continuous support, and in particular Christopher Creuztig, Klaus Drescher, Ralf Hillebrand, and Stefan Wehmeier.

References

[CGG+88] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton Leong, Michael B. Monagan, and Stephen M. Watt. *Maple Reference Manual.* WATCOM Publications Limited, 415 Philip St, Waterloo, Ontario N2L 3X2, Canada, fifth edition, 1988.

[DDZ98] Alain Denise, Isabelle Dutour, and Paul Zimmermann. Cs: a mupad package for counting and randomly generating combinatorial structures. *Proceedings of FPSAC'98*, pages 195–204, 1998.

[FZVC94] Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoret. Comput. Sci.*, 132(1-2):1–35, 1994.

[GAP99] The GAP Group, Aachen, St Andrews. *GAP – Groups, Algorithms, and Programming, Version 4.1*, 1999.

[HNJY02] Florent Hivert, Jean-Christophe Novelli, and Thibon Jean-Yves. An analogue of the plactic monoid for binary search trees. *Comptes Rendus de l'Académie des Sciences*, 2002.

[HT02] Florent Hivert and Nicolas M. Thiéry. Deformation of symmetric functions and the rational steenrod algebra. 2002. accepted.

[Ker91] Adalbert Kerber. *Algebraic combinatorics via finite group actions.* Bibliographisches Institut, Mannheim, 1991.

[KKL] Adalbert Kerber, Axel Kohnert, and Alain Lascoux. SYMMETRICA, an object oriented computer-algebra system for the symmetric group.

[KS98] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms; Generation, Enumeration and Search.* Discrete mathematics and its applications. CRC Press, 1998.

[Mac95] I. G. Macdonald. *Symmetric functions and Hall polynomials*, volume 166 of *Oxford mathematical monographs*. Clarendon, second edition, 1995.

[Pro99] V. Prosper. *Combinatoire des polynômes multivariés*. PhD thesis, IGM, Université de Marne la Vallée, 1999.

[Rot75] Gian-Carlo Rota. *Finite operator calculus*. Academic press, 1975.

[SW86] Dennis Stanton and Dennis White. *Constructive combinatorics*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1986.

[The96] The MuPAD Group, Benno Fuchssteiner et al. *MuPAD User's Manual - MuPAD Version 1.2.2*. John Wiley and sons, Chichester, New York, first edition, march 1996. includes a CD for Apple Macintosh and UNIX.

[Thi99] Nicolas M. Thiéry. *Invariants algébriques de graphes et reconstruction; une étude expérimentale*. PhD thesis, Université Lyon I, June 1999. N° d'ordre: 167-99.

[Thi00] Nicolas M. Thiéry. Algebraic invariants of graphs: a study based on computer exploration. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 34(3):9–20, September 2000.

[Thi01] Nicolas M. Thiéry. Computing minimal generating sets of invariant rings of permutation groups with SAGBI-Gröbner basis. In *Discrete models: combinatorics, computation, and geometry (Paris, 2001)*, Discrete Math. Theor. Comput. Sci. Proc., AA, pages 315–328 (electronic). Maison Inform. Math. Discrèt. (MIMD), Paris, 2001.

[Vei98] S. Veigneau. ACE, an Algebraic Combinatorics Environment for the computer algebra system MAPLE: User's Reference Manual, Version 3.0. Report 98–11, IGM, 1998.

[Zim94] Paul Zimmermann. Gaia: A package for the random generation of combinatorial structures. *The Maple Technical Newsletter*, 1(1), 1994.