CS: a MuPAD Package for Counting and Randomly Generating Combinatorial Structures

Alain Denise
LRI, Université Paris-Sud
Alain.Denise@lri.fr

Isabelle Dutour
INRIA Lorraine
Isabelle.Dutour@loria.fr

Paul Zimmermann
INRIA Lorraine
Paul.Zimmermann@loria.fr

January 21, 1998

Abstract

We present a new computer algebra package which permits to count and to generate combinatorial structures of various types, provided that these structures can be described by a specification, as defined in [7].

Résumé

Nous présentons un nouveau module de calcul formel dédié au dénombrement et à la génération aléatoire uniforme de structures combinatoires décomposables.

1 What is CS ?

CS is a computer algebra package devoted to the handling of combinatorial structures. Its main features are the following: given a combinatorial specification of a class of *decomposable* structures (in the sense of [7]), CS is able to count and uniformly draw at random the structures of any given size n. It can also give some properties of the associated generating series, like recurrences and differential equations.

CS, like COMBSTRUCT (previously known as Gaïa [10]), another package developed by one of the authors, is mainly based on the theory developed in [7] and [6]. But, in addition to the fact that CS is a MuPAD package while COMBSTRUCT is a Maple package, CS presents some more advanced features: at the time of submission, CS offers the same fonctionalities than COMBSTRUCT offers concerning unlabelled structures (excepted for cycles and some conditions on cardinality of sets and sequences), plus the following:

- Given any decomposable class, CS is able to automatically generate the source of a C program for almost uniform random generation. Then this program can be compiled and used totally independently from MuPAD. You are then able to get the approximate counting or draw at random one or several of its elements of desired size, much more efficiently than with a Maple or a MuPAD procedure. This is the first implementation of one of the algorithms given in [4].

- When the generating series of a decomposable class is holonomic, CS is able to compute, using Gröbner basis calculation and Gaussian elimination, the minimal order recurrence that its coefficients satisfy. This fairly improves the complexity of counting and enables generating larger structures.

At the time of submission, CS only deals with unlabelled structures. The CS package is currently being developed and we plan to add the labelled case and several other new features; Section 5 describes some of them.

2 How to install CS on your computer

First, you must have installed the MuPAD computer algebra system on your computer. This software is distributed for free (but not in public domain: a registration is required). It can be downloaded at the following address:

```
http://www.mupad.de
```

MuPAD is available on several computer systems and the way to install it on each system is precisely described.

Once you have MuPAD on your computer, you can download the CS package (one ASCII file) at the address:

```
http://www.loria.fr/~dutour/CS/cs.mu
```

Now you just have to run MuPAD and to read CS by typing, after the prompt "`>>`",

```
read("cs.mu"):
```

3 Counting and Generating

A *specification* of a class of combinatorial structures, as defined in [7], is a set of productions made from basic objects (atoms) (**Epsilon** and **Z** of size 0 and 1 respectively) and from constructions (**Union** for disjoint unions, **Prod** for products, **Sequence** for sequences, **Set** for sets (labelled case) or multisets (unlabelled case)).

Here is the MuPAD instruction which creates the specification of plane trees:

```
>> specPT:={T=Prod(Z,Sequence(T))};

                  {T = Prod(Z, Sequence(T))}
```

In other words, a plane tree is a product of a root vertex (the atom `Z`) and a sequence (possibly empty) of plane subtrees. To count the number of plane trees of size, say 100, use the `count` procedure of the CS package:

```
>> cs::count([T,specPT],size=100);

        227508830794229349661819540395688853956041682601541047340
```

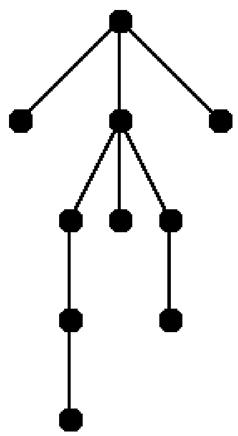
We can verify that the first coefficients of the generating series correspond to the first Catalan numbers:

```
>> cs::count([T,specPT],size=i)$i=1..10;

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

Now let us draw a random plane tree with 10 vertices:

```
>> cs::draw([T,specPT],size=10);

Prod(Z, Prod(Prod(Z, Epsilon), Prod(Prod(Z, Prod(Prod(Z,

   Prod(Prod(Z, Prod(Prod(Z, Epsilon), Epsilon)), Epsilon)),

   Prod(Prod(Z, Epsilon), Prod(Prod(Z,

   Prod(Prod(Z, Epsilon), Epsilon)), Epsilon)))),

   Prod(Prod(Z, Epsilon), Epsilon))))
```

This description corresponds to the following tree:



Remark: Such a figure can be drawn almost directly from the output of the `draw` procedure by using CalICo [3] or CGraph [2], two programs dedicated to the manipulation and visualization of combinatorial structures. This figure was drawn with CalICo.

Here is another example of a specification, using an argument of cardinality; `P` represents the integer partitions and `N` the integers greater or equal to 1:

```
>> specPART:={P=Set(N),N=Sequence(Z,card>=1)};

                  {N = Sequence(Z, 1 <= card), P = Set(N)}

>> cs::draw([P,specPART],size=5);

   Prod(Prod(Z, Epsilon), Prod(Prod(Z, Prod(Z,

   Prod(Z, Prod(Z, Epsilon)))), Epsilon))
```

The `count` and `draw` procedures both call the `compile` procedure. This last one produces, given the specification, the recurrences which will be used in order to count and generate the structures. The variable `HOLONOMIC`, when switched to the value `TRUE`, tells CS to test, when compiling a class of structures, if its generating function is holonomic or not. If yes, CS finds the simplest linear recurrence to count the structures, so that the counting runs more quickly. (The default value of `HOLONOMIC` is `FALSE`). To illustrate this, compare the time needed to generate Motzkin trees of size 100, when compiling with `HOLONOMIC` equal to `FALSE` or `TRUE` (time in ms):

```
>> specMT:={M=Union(Z,Prod(Z,M),Prod(Z,M,M))};

                  {M = Union(Z, Prod(Z, M), Prod(Z, M, M))}


>> cs::HOLONOMIC:=FALSE:
>> st:=time(): cs::compile(specMT): time()-st;

                                    2250


>> st:=time(): cs::count([M,specMT],size=100); time()-st;

                 2494785789912243786801425614600100304678115800

                                   539720


>> cs::HOLONOMIC:=TRUE:
>> st:=time(): cs::compile(specMT): time()-st;

                                    5740


>> st:=time(): cs::count([M,specMT],size=100); time()-st;

                 2494785789912243786801425614600100304678115800

                                     670
```

In the second case, the call to `compile` is longer, due to the computation using Gröbner basis and Gaussian elimination; but once finished, the counting and random generation are much more efficient. Before compiling with `HOLONOMIC=TRUE`, you can call the MuPAD function `setuserinfo` in order to see the operations involved by the calculation of the linear recurrences:

```
>> cs::HOLONOMIC:=TRUE:
>> setuserinfo(Any,1):
>> cs::compile(specMT);
```

```
...
standard form is: , {T2 = Prod(T1, M), T1 = Prod(Z, M), Z = Atom, M = Unio\
n(Z, T1, T2)}
...
polynomials are: , [T2*(-1) + M*T1, T1*(-1) + M*z, M*(-1) + z + T1 + T2]
auto reduces list of 3 polynomials
groebner basis is: , [M^2 + M*z^(-1)*(z + (-1)) + 1, z + T2 + M*(z + (-1))\
, T1 + M*z*(-1)]
perform (ordinary) Gaussian elimination
algeq is: , T2*(-1) + z*T2*2 + z^3 + z*T2^2 + z^2*T2
perform (ordinary) Gaussian elimination
diffeq is: , T2(z) + z*T2(z)*(-1) + z*D(T2)(z) + z^3*(-4) + z^2*T2(z)*(-1)\
 + z^3*T2(z)*(-3) + z^2*D(T2)(z)*(-3) + z^3*D(T2)(z)*(-1) + z^4*D(T2)(z)*3
recurrence for , T2,  is: , u(n) + n*u(n) + u(n + (-1))*(-1) + u(n + (-2))\
*(-1) + u(n + (-3))*(-3) + (n + (-1))*u(n + (-1))*(-3) + (n + (-2))*u(n + \
(-2))*(-1) + (n + (-3))*u(n + (-3))*3
...
perform (ordinary) Gaussian elimination
algeq is: , T1*(-1) + z*T1 + z^2 + T1^2
perform (ordinary) Gaussian elimination
diffeq is: , z*2 + T1(z)*(-1) + D(T1)(z)*(-1) + z*T1(z)*(-3) + z*D(T1)(z)*\
2 + z^2*D(T1)(z)*3
recurrence for , T1,  is: , u(n)*(-1) + n*u(n)*2 + u(n + (-1))*(-3) + (n +\
 (-1))*u(n + (-1))*3 + (n + 1)*u(n + 1)*(-1)

                                   TRUE
```

The `compile` procedure allows additional arguments in order to create the source file of a C program that allows to generate more quickly one or several structures of a given size. This program can be compiled and run either from or out of MuPAD. The following example shows how to process the first possibility.

```
>> cs::compile(specMT,target=C,file="MotzkinTree.c",main=M);

                                   TRUE

>> system("cc -Dsun4 MotzkinTree.c -lm; a.out 4");

                         Prod(Z, Prod(Prod(Z, Z), Z))
```

The binary file `a.out` can take two arguments. The first is the size of the structure (n) and the second is the number of structures you want to generate. If this second argument is not present, the program generates only one structure, and if it is equal to 0, it gives the approximate number of structures of size n:

```
>> system("a.out 1000 0");

         2.0408143590960460*10^472..2.0408143590972756*10^472
```

In fact, the result consists in an interval; we can guarantee that the exact number of structures lies in this interval, because the program does respect the IEEE specifications concerning floating-point calculations. This can be easily verified for our example:

```
>> cs::count([M,specMT],size=1000);

204081435909723394323514347945305410637315566977592672298454731935657974 75\
190764578929189419806192270968498827932500196689732989494882093156254684 90\
845464149184562077955733605472889039145344631960432462433436041215113489 77\
643373370150067363376992424517680239368433045926601010027275549662287508 66\
899189589142612444110305960895761119433635559186014525321732540295556045 85\
912182122935465188225509404058831423800392236484440452873562422240288297 78\
7354959394301055653483556 6509

>> length(%);
                                   473
```

The C program allows to manipulate very large numbers (see above). Normally, it can be compiled on all machines. Presently, the supported architectures are SunOs (-Dsunos), Solaris (-Dsun4), DecAlpha (-Dalpha), IRIX (-DIRIX64).

In the current version of CS, the C program only performs *almost uniform* random generation, because the coefficients used in the generation procedures are computed in floating-point arithmetic. (However, since the procedures are numerically stable, the rounding errors involved by floating-point arithmetic do not strongly affect the quasi-uniformity of generation; for details, see [4]). A future version of CS will allow to produce exactly uniform generators in C language (see Section 5).

4 Equations and Recurrences

The CS package provides some procedures in order to handle generating series. They implement (in MuPAD) some fonctionalities of the GFUN package [9]. These procedures are used in `compile` but can also be called independently.

The procedure `speciftoalgeq` returns an algebraic equation that the generating series associated to the given specification satisfies, provided that such an equation exists:

```
>> eq:=cs::speciftoalgeq(specMT,M);
                                          2
                          z - M + M z + M  z
```

(At first glance, this does not look like an equation, but this is to be read as $z - M + Mz + M^2 z = 0$.)

The procedure `algeqtodiffeq` converts this equation into a differential equation, while `diffeqtorec` gives the recurrence on the coefficients involved by the differential equation:

```
>> deq:=cs::algeqtodiffeq(eq,M(z));
                                            2               3
    2 z - M(z) + z M(z) - z D(M)(z) + 2 z  D(M)(z) + 3 z  D(M)(z)
```

```
>> r:=cs::diffeqtorec(deq, M(z), v(n));

    v(n - 1) - v(n) - n v(n) + 2 (n - 1) v(n - 1) + 3 (n - 2) v(n - 2)
```

We can resume all these steps in one unique call:

```
>> r:=cs::speciftorec(specMT,M,v(n));

    v(n - 1) - v(n) - n v(n) + 2 (n - 1) v(n - 1) + 3 (n - 2) v(n - 2)
```

5 Future Features

The CS package is presently at the beginning of its development (the version number, at the time of submission, is 0.8). Our aim is to add, within the next months, several new features. Here are some of them.

- A future version of CS will handle labelled structures as well as unlabelled ones.
- One of the authors has developed a Maple package, named qAlGO, mainly based on the notion of object grammars [5], and that leads to enumeration and random generation of objects according to non algebraic parameters. The packages qAlGO and COMBSTRUCT complement each others. We plan to integrate also the qAlGO technics in CS.
- Recently, two of the authors investigated a method, based on a strategy of "lazy evaluation", which fairly improves the average complexity of the exactly uniform random generation of decomposable structures [4]. We plan to use this method in CS in order to get more efficient generation procedures.
- In the current version of CS, the C programs written by the `draw` procedure only perform almost uniform random generation. We intend to use a multiprecision mathematical library, like PARI [1] or GMP [8], so that CS can write true random generators in C language.

References

[1] C. Batut, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*, January 1995. URL `ftp://megrez.math.u-bordeaux.fr/pub/pari`.

[2] François Bertault. *Génération et tracé de structures décomposables.* PhD thesis, Université Henri Poincaré Nancy 1, September 1997.

[3] M. Delest, J.M. Fédou, G. Melançon, and N. Rouillon. Computation and images in combinatorics. In RIACA Amsterdam, editor, *HISC book*. Springer Verlag, To appear.

[4] A. Denise and P. Zimmermann. Uniform random generation of decomposable structures using floating-point arithmetic. Technical Report 3242, INRIA, 1997. Submitted to Theoretical Computer Science.

[5] I. Dutour and J.M. Fédou. Object grammars and random generation. Technical Report 1165-97, LaBRI, Université Bordeaux I, 1997. Submitted to CompSciNet: Discrete Mathematics and Theoretical Computer Science.

[6] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus of random generation: Unlabelled structures. In preparation.

[7] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132:1–35, 1994.

[8] Torbjörn Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, June 1996.

[9] B. Salvy and P. Zimmermann. Gfun: A Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, June 1994.

[10] P. Zimmermann. Gaïa: a package for the random generation of combinatorial structures. *MapleTech*, 1(1):38–46, 1994.