1 The series package

1.1 Intro

The series FORM package tries to capture parts of the Mathematica `Series` functionality. It implements basic operations on a power series in one (and at some point maybe even more than one) variable (e.g. ϵ) up to some cut-off power ϵ^{cut}. The supported operations include basic replacements, multiplication of series and computing a series to some power (including e.g. the inverse and the square root of the series), where the exponent can be another series. At the moment, several of these features *are turned of*, mostly for performance reasons. It is recommended to use their standard FORM counterparts instead.

1.2 Quick start

This chapter is only for impatient people who do not like reading manuals. The rest may want to continue with Chapter 1.3.

```
*include series package
#include- series.h

S ep;
L foo=1-ep;

*define a series variable and cut
#call series(ep,3)

*invert foo & save result in bar
#call invert(foo,bar)
drop foo,bar;

*initialise operations on functions
#call init(3)

L foo=exp(1-ep)+exp(1+ep);

*expand exponential  function
#call expand(exp)

.end
```

1.3 Usage

To make the series procedures available, include the header file in your FORM program:

```
#include- series.h
```

Make sure that `series.h` resides at some location where FORM will find it, preferably in one of the directories defined by the environment variable FORMPATH (see Chapter 1 of the FORM reference manual). The package tries to interfer as few as possible with the rest of your program, but there are some caveats:

- `exp` and `log` are sometimes defined as commuting functions inside some of the procedures. Do not misuse these names.
- Some procedure names are reserved. See also Chapter 2.1 on namespaces.
- Refrain from using any symbols, expressions, etc. of the form `[series::`*name*`]`. They are reserved. Dollar variables `$series`*name* are also a bad idea.

There are two modes of usage: it is possible to either expand *expressions* (defined with the FORM instructions `local` or `global`) or *functions.* These modes are very different in nature, both in their usage and in their internal implementations.

The first mode, operating on expressions, is more mature and probably signigicantly faster. Use this mode, if you can. First, you will want to call the procedure `series`, e.g.

```
#call series(ep,7)
```

This tells the package to treat all expressions as series in `ep` up to power 7. After that, you can use one of the procedures `log, exp, power` or `invert` with two expression names as arguments to perform the corresponding operation on the first argument and save the result in the second argument. See the Chapter 2 on procedures to find out which procedure does what.

The second mode, operating on functions, is not tested as well and probably a lot slower. In order to use it, you have to call the procedure `init` first:

```
#call init(9)
```

Its argument is the highest number of terms in a series that will be computed. If it is set too low, some strange functions can appear in your expressions. Make sure that the file `partition.tbl` is in your FORMPATH; this mode will not work if it is not found by FORM.

Like in the first mode, you have to call the `series` procedure first. Afterwards it is most convenient to call `expand` to expand a function.

2 Procedures

2.0.1 series

Before you use any other procedure from the series package, you have to define a global series variable and a cut-off. This is done with the procedure `series`. The first parameter is the series variable, the second one is the highest power of that variable. For example

```
#call series(ep,7)
```

just tells all following procedures that the expressions are series in `ep` up to power 7. Note that the `series` procedure itself does not change expressions or do anything visible.

2.0.2 power

This procedure computes an expression (first argument) to some power (second argument) and stores the result in some other expression (third argument). The first two arguments should be expressions and will be treated as series. This FORM program will compute $(1+ep)^{1+ep}$ up to the power ep^3 and print the result:

```
#include- series.h
S ep;
L [1]=1+ep;
L [2]=[1];
#call series(ep,3)
#call power([1],[2],[(1+ep)^(1+ep)])
print [(1+ep)^(1+ep)];
.end
```

2.0.3 log

`log` computes the logarithm of the first argument (an expression) and stores it in an expression named like the second argument. This program will compute $\log(1+ep)$ up to power ep^3:

```
#include- series.h
S ep;
L [1]=1+ep;
#call series(ep,3)
#call log([1],[log(1+ep)])
print [log(1+ep)];
.end
```

2.0.4 exp

Like `log` (section 2.0.3), but computes the exponential function. Compute e^{ep} up to order ep^3:

```
#include- series.h
S ep;
L [1]=ep;
#call series(ep,3)
#call exp([1],[exp(ep)])
print [exp(ep)];
.end
```

2.0.5 invert

This procedure inverts a series expression (first argument) and stores it in the expression given as the second argument. Thus

```
#call  invert(series,result)
```

is equivalent to

```
L [-1]=-1;
#call  invert(series,[-1],result)
drop [-1];
```

but more readable and faster.

2.0.6 init

This procedure initialises operations on functions. If you want to use `expand` or any of the `*function` procedures, you have to call this procedure first. Its argument is the highest possible amount of summands in a series. Depending on the appearance of poles this can be higher or even lower than your cutting power. (E.g. for a $1/\epsilon$ pole and a cut of 5 you shold use `#call init(6)`)

2.0.7 expand

Tries to expand the function that is given as an argument. "Known" functions are `exp, log, ln, pow, power, den` and `deno` (case sensitive). If your favourite logarithm function has the name `Log`, try the procedure `logfunction` instead. This example will compute the inverse of $1 + ep$:

```
#include- series.h
#call init(3)
S ep;
CF den;
L [1]=den(1+ep);
#call series(ep,3)
#call expand(den)
print [1];
.end
```

2.0.8 powerfunction

This procedure also computes a series to some power, but works on functions instead of expressions. The argument of `powerfunction` is the name of the function, the function's first argument is the base and the second argument is the exponent. This example will again compute $(1 + ep)^{1+ep}$:

```
#include- series.h
#call init(3)
```

```
S ep;
CF pow;
L [1]=pow(1+ep,1+ep);
#call series(ep,3)
#call powerfunction(pow)
print [1];
.end
```

Note that this procedure is probably *much slower* than its counterpart which works on expressions. Use it with care and only when needed.

2.0.9 logfunction

This procedure also computes the logarithm of a series, but works on functions instead of expressions. The argument of `logfunction` is the name of the logarithm function. This example will again compute $\log(1+ep)$:

```
#include- series.h
#call init(3)
S ep;
CF log;
L [1]=log(1+ep);
#call series(ep,3)
#call logfunction(log)
print [1];
.end
```

Note that this procedure is probably *much slower* than its counterpart which works on expressions. Use it with care and only when needed.

2.0.10 expfunction

This procedure also computes the exponential of a series, but works on functions instead of expressions. The argument of `expfunction` is the name of the exponential function. This example will again compute e^{1+ep}:

```
#include- series.h
#call init(3)
S ep;
CF exp;
L [1]=exp(1+ep);
#call series(ep,3)
#call expfunction(exp)
print [1];
.end
```

Note that this procedure is probably *much slower* than its counterpart which works on expressions. Use it with care and only when needed.

2.0.11 invertfunction

This procedure also computes the inverse of a series, but works on functions instead of expressions. The argument of `invertfunction` is the name of the function whose argument is to be inverted. This example will compute $1/(1+ep)$:

```
#include- series.h
#call init(3)
S ep;
CF den;
L [1]=den(1+ep);
#call series(ep,3)
#call invertfunction(den)
print [1];
.end
```

Note that this procedure is probably *much slower* than its counterpart which works on expressions. Use it with care and only when needed.

2.0.12 parallel

Usually parallel processing is turned of for modules containing procedures which expand functions (`expand, *function`) because of dollar variables. If you need parallel computing call the `parallel` procedure *directly* before the end of the module.

2.0.13 createtable

For internal use only. Creates a table for storing expressions.

2.0.14 toseries

For internal use only. Puts an expression into a previously defined table.

2.0.15 partition

For internal use only. Computes partitions of integer numbers.

2.1 Namespaces

Unfortunately, FORM has no clean interface for packages: There is no such thing as namespaces and scoping is (almost) nonexistant. The `series` package tries to work around this by assigning very complicated names for its expressions and symbols, which hopeful no sane person would ever use in his or her FORM program.

Still, there is a problem with the names of procedures which might clash with user-defined ones (did I mention that FORM doesn't know overloading either?). To work around this, the `series` package has very basic support for namespaces. If the preprocessor variable `NAMESPACE` is defined *before* the `series` package is included, all procedures will have the value of `NAMESPACE` prepended to their name:

```
*import the series procedures into the namespace "series"
* DO NOT undefine NAMESPACE
#define NAMESPACE "series"
#include- series.h

L foo=1+ep;
*"series" now becomes "seriesseries"
#call seriesseries(ep,3)

*"exp" becomes "seriesexp"
#call seriesexp(foo,bar)
print bar;
.end
```

2.2 Implementation

This section covers implementation details. If you just want to use the package, you can savely stop reading here.

2.2.1 Generalities

We usually want to compute

$$f\left(\sum_{i=0}^{\infty} a_i \epsilon^i\right) = \sum_{n=0}^{\infty} b_i \epsilon^i \,. \tag{1}$$

and express the coefficients b_i in terms of the known a_j. We can distinguish between two kinds of formulations: the *recursive* representation, where b_i is a function of a_j and b_k with $k < i$ and the *explicit* representation, where b_i is a function of the a_j only. We will usually try to find and use a recursive representation because it involves a lower number of terms. At the moment, the recursive representation is used for operations on expressions; for operation on function arguments (`expfunction, logfunction,...`) I have not yet succeeded in implementing it and use the explicit representation instead.

The explicit representations make heavy use of partitions. $\mathcal{P}(n)$ is the set of partitions of n i.e. for any Partition $P \in \mathcal{P}(n)$ the sum over all elements of P is n. $|P|$ is the cardinality of P. $\mathcal{M}_P(l)$ is the multiplicity of l in a partition P.

A simple example:

$$\begin{aligned}\mathcal{P}(4) =&\{(1,1,1,1),\ (1,1,2),\ (1,3),\ (2,2),\ (4)\}\\ |(1,1,2)| =&3\\ \mathcal{M}_{(1,1,2)}(1) =&2\\ \mathcal{M}_{(1,1,2)}(2) =&1\end{aligned}$$

This means e.g.

$$\sum_{P\in\mathcal{P}(4)}\left(\prod_{l\in P}\frac{c_l^{\mathcal{M}_P(l)}}{\mathcal{M}_P(l)!}\right) = \frac{c_1^4}{4!} + \frac{c_1^2}{2!}\frac{c_2^1}{1!} + \frac{c_1^1}{1!}\frac{c_3^1}{1!} + \frac{c_2^2}{2!} + \frac{c_4^1}{1!}\,. \tag{2}$$

2.2.2 Exponentiation

Exponentiation is required for the evaluation of terms of the form series_1$^{\text{series\_2}}$. The recursive representation I use is

$$b_0 =1 \tag{3}$$

$$b_n =\sum_{i=1}^{n}\frac{i}{n}a_i b_{n-i} \tag{4}$$

An explicit representation of the b_i is

$$b_n = \sum_{P\in\mathcal{P}(n)}\left(\prod_{l\in P}\frac{c_l^{\mathcal{M}_P(l)}}{\mathcal{M}_P(l)!}\right) \tag{5}$$

For the definitions of the partition set $\mathcal{P}$ and the multiplicity $\mathcal{M}_P$ see the section 2.2.1 on exponentials. At the moment, however, the function version just splits the exponential function into a product and iteratively replaces

$$exp(a_n\epsilon^n) = \sum_{i=1}^{\infty} a_n^i \epsilon^{n\cdot i} \tag{6}$$

up to the required order.

2.2.3 Logarithms

Logarithms are also required to compute complicated powers. Here I used

$$\log\left(\sum_{i=k_0}^{\infty} c_i\epsilon^i\right) =\log\left(a_{k_0}\epsilon^{k_0}\right) + \log\left(1+\sum_{i=1}^{\infty}a_i\epsilon^i\right) \tag{7}$$

$$\log\left(1+\sum_{i=1}^{\infty}a_i\epsilon^i\right) =\sum_{n=1}^{\infty}b_n\epsilon^n\,. \tag{8}$$

I try to use the following recursive representation of the b_n:

$$b_1 = a_1 \tag{9}$$

$$b_n = a_n - \sum_{i=1}^{n-1} \frac{i}{n} a_{n-i} b_i \tag{10}$$

An explicit representation of the b_n is

$$b_n = \left[\sum_{P \in \mathcal{P}(n)} (-1)^{|P|-1} (|P|-1)! \left(\prod_{l \in P} \frac{c_l^{\mathcal{M}_P(l)}}{\mathcal{M}_P(l)!} \right) \right] \tag{11}$$

For the definitions of the partition set $\mathcal{P}$ and the multiplicity $\mathcal{M}_P$ see the section 2.2.1 on exponentials.

2.2.4 Powers

Powers y of series x are computed using the formula

$$x^y = \exp(y * log(x))\,. \tag{12}$$

2.2.5 Inverse

Here I use

$$\left(\sum_{i=k_0}^{\infty} c_i \epsilon^i \right) = c_{k_0} \epsilon^{k_0} \left(1 + \sum_{i=1}^{\infty} a_i \epsilon^i \right) \tag{13}$$

$$\left(1 + \sum_{i=1}^{\infty} a_i \epsilon^i \right)^{-1} = \sum_{i=0}^{\infty} b_i \epsilon^i \tag{14}$$

with the recursive representation (for expressions)

$$b_0 = 1 \tag{15}$$

$$b_n = - \sum_{i=0}^{n-1} a_{n-1} b_i \tag{16}$$

and the explicit representation (for functions)

$$b_n = \left[\sum_{P \in \mathcal{P}(n)} (-1)^{|P|} |P|! \left(\prod_{l \in P} \frac{c_l^{\mathcal{M}_P(l)}}{\mathcal{M}_P(l)!} \right) \right] \tag{17}$$

For the definitions of the partition set $\mathcal{P}$ and the multiplicity $\mathcal{M}_P$ see the section 2.2.1 on exponentials.

2.3 Deprecated & development

This is a collection of procedures that are not included in the distribution of the package. They are mostly deprecated or much too early in the development stage.

2.3.1 contractpowers

This is a special procedure for simplifying powers like

```
(x)^(y/2)*(x)^(y/2)
```

FORM does not do this automatically - if you encounter such terms, use

```
#call contractpowers
```

While it is certainly useful, this procedure no longer fits with the rest of the package; it might be included in a different future package.

2.3.2 Simple Powers

This procedure computes a series to a simple (i.e. non-series) power. Here the formula

$$\left(\sum_{i=0}^{\infty} c_i \epsilon^i\right)^j = \sum_{n=0}^{\infty}\left[\sum_{P\in\mathcal{P}(n)}\left(\prod_{k=0}^{|P|-1}(j-k)\right)\left(\prod_{l\in P}\frac{c_l^{\mathcal{M}_P(l)}}{\mathcal{M}_P(l)!}\right)\right]\epsilon^n \tag{18}$$

is used. For the definitions of the partition set $\mathcal{P}$ and the multiplicity $\mathcal{M}_P$ see the section 2.0.4 on exponentials. This procedure has performance issues (is slower than `power`) and hence is not included.

2.3.3 Identify

This procedure is like the usual FORM `id` statement, but does not compute any terms that will be thrown away. Still, it is too slow and not included in the package.

2.3.4 Multiply

This procedure is like the usual FORM `multiply` statement, but does not compute any terms that will be thrown away. Still, it is too slow and not included in the package.