`datatypes` — Basic MuPAD data types

Table of contents

`DOM_DOMAIN` – the data type of data types

Each MuPAD object is of a unique data type. Since a data type is a MuPAD object, too, it must itself have a data type; the data type comprising all data types (including itself) is `DOM_DOMAIN`.

There are two kinds of elements of `DOM_DOMAIN`: data types of the kernel, and data types defined in the library or by the user (*domains*). Objects that have a data type of the latter kind are called *domain elements*.

A data type has the same internal structure as a table; its entries are called *slots*. One particular slot is the *key*; no two different data types can have the same key. Most of the other slots determine how arguments of that data type are handled by functions.

Once a user-defined domain has been constructed, it cannot be destroyed.

Construction:

The names of the data types provided by the MuPAD kernel are of the form `DOM_XXX`, such us `DOM_ARRAY`, `DOM_IDENT`, `DOM_INT`, `DOM_LIST`, `DOM_TABLE` etc.

You can create further data types using the function `newDomain` (cf. example 1) or via the keyword `domain` (cf. example 3).

You can also create new data types by calling a *domain constructor*. Various pre-defined domain constructors can be found in the library `Dom`. You can also define your own domain constructors using the keyword `domain`. Cf. example 2.

The domain type (data type) of any MuPAD object can be queried by the function `domtype`.

Important Operations:

- You can obtain the slots of a domain using `slot`. The function `slot` can also be used on the left hand side of an assignment to define new slots, or to re-define existing slots. Use `delete` to delete slots.

Result of Evaluation: Evaluating an object of the domain type `DOM_DOMAIN` returns itself.

Function Call: When called as a function, the data type creates a new object of this data type out of the arguments of the call. E.g., the call `DOM_LIST(1, 2, x)` generates the list `[1, 2, x]` of domain type `DOM_LIST` (although, in this case, you probably prefer to type in `[1, 2, x]` directly which results in the same object). It depends on the particular type which arguments are admitted here.

In the case of a domain, the `"new"` method of that domain is called.

Operands: A data type consists of an arbitrary number of equations (objects of type `"equal"`). If `a = b` is among these equations, we say that the *slot* `a` of the data type equals `b`. By convention, `a` is usually a string. Each domain has at least one slot indexed by `"key"`.

Example 1. Our first example stems from ethnology: some languages in Polynesia do not have words for numbers greater than three; every integer greater than three is denoted by the word "many". Hence two plus two does not equal four but "many". We are going to implement a domain for this kind of integers; in other words, we are going to implement a data type for the finite set `{1, 2, 3, many}`.

```
>> S := newDomain("Polynesian integer")

                    Polynesian integer
```

At this point, we have defined a new data type: a MuPAD object can be a Polynesian integer now. No operations are available yet; the domain consists of its key only:

```
>> op(S)

               "key" = "Polynesian integer"
```

Even though there are no methods for input and output of domain elements yet, Polynesian integers can be entered and displayed right now. You have to use the function `new` for defining domain elements:

```
>> x := new(S, 5)

               new(Polynesian integer, 5)
```

Now, `x` is a Polynesian integer:

```
>> type(x)

                    Polynesian integer
```

Of course, MuPAD cannot know what a Polynesian integer stands for and what its internal structure should be. The arguments of the call to the function `new` are just stored as the zeroth, first, etc. operand of the domain element, without checking them. You may call `new` with as many arguments as you want:

```
>> new(S, 1, 2, 3, 4); op(%)
```

```
                  new(Polynesian integer, 1, 2, 3, 4)

                              1, 2, 3, 4
```

`new` cannot know that Polynesian integers should have exactly one operand and that we want `5` to be replaced by `many`. To achieve this, we implement our own method `"new"`; this also allows us to check the argument. We have one more problem: domain methods should refer to the domain; but they should not depend on the fact that the domain is currently stored in `S`. For this purpose, MuPAD has a special local variable `dom` that always refers to the domain a procedure belongs to:

```
>> S::new :=
   proc(i : Type::PosInt)
   begin
     if args(0) <> 1 then
       error("There must be exactly one argument")
     end_if;
     if i > 3 then
       new(dom, hold(many))
     else
       new(dom, i)
     end_if
   end_proc:
```

A function call to the domain such as `S(5)` now implicitly calls the `"new"` method:

```
>> S(5)

                  new(Polynesian integer, many)

>> S("nonsense")

 Error: Wrong type of 1. argument (type 'Type::PosInt' expected\
 ,
        got argument '"nonsense"');
 during evaluation of 'S::new'
```

In the next step, we define our own output method. A Polynesian integer `i`, say, shall not be printed as `new(Polynesian integer, i)`, only its internal value `1`, `2`, `3`, or `many` shall appear on the screen. Note that this value is the first operand of the data structure:

```
>> S::print :=
   proc(x)
   begin
     op(x, 1)
   end_proc:
   S(1), S(2), S(3), S(4), S(5)
```

```
1, 2, 3, many, many
```

By now, the input and output of elements of `S` have been defined. It remains to define how the functions and operators of MuPAD should react to Polynesian integers. This is done by *overloading* them. However, it is not necessary to overload each of the thousands of functions of MuPAD; for some of them, the default behavior is acceptable. For example, expression manipulation functions leave domain elements unaltered:

```
>> x := S(5): expand(x), simplify(x), combine(x); delete x:

                    many, many, many
```

Arithmetical operations handle domain elements like identifiers; they automatically apply the associative and commutative law for addition and multiplication:

```
>> (S(3) + S(2)) + S(4)

                      many + 2 + 3
```

In our case, this is not what we want. So we have to overload the operator `+`. Operators are overloaded by overloading the corresponding "underline-functions"; hence, we have to write a method `"_plus"`:

```
>> S::_plus :=
   proc()
   local argv;
   begin
     argv := map([args()], op, 1);
     if has(argv, hold(many)) then
        new(dom, hold(many))
     else
        dom(_plus(op(argv)))
     end_if
   end_proc:
```

Now, the sum of Polynesian integers calls this slot:

```
>> S(1) + S(2), S(2) + S(3) + S(7)

                         3, many
```

Deleting the identifier `S` does not destroy our domain. It can still be reconstructed using `newDomain`.

```
>> delete S: op(newDomain("Polynesian integer"))

 "_plus" = proc S::_plus() ... end,

    "print" = proc S::print(x) ... end,

    "new" = proc S::new(i) ... end, "key" = "Polynesian integer"
```

Example 2. We could now give a similar example for more advanced Polynesian mathematics with numbers up to ten, say. This leads to the question whether it is necessary to enter all the code again and again whenever we decide to count a bit farther. It is not; this is one of the advantages of *domain constructors*. A domain constructor may be regarded as a function that returns a domain depending on some input parameters. It has several additional features. Firstly, the additional keywords `category` and `axiom` are available for specifying the mathematical structure of the domain; in our case, we have the structure of a commutative semigroup where different domain elements have different mathematical meanings (we call this a domain with a canonical representation). Secondly, an initialization part may be defined that is executed exactly once for every domain returned by the constructor; it should at least check the parameters passed to the constructor. Each domain created in such a way may inherit methods from other domains, and it must at least inherit the methods of `Dom::BaseDomain`. You find more detailed information in the domains reference.

```
>> domain CountingUpTo(n : Type::PosInt)

   inherits Dom::BaseDomain;
   category Cat::AbelianSemiGroup;
   axiom Ax::canonicalRep;

   new := proc(x : Type::PosInt)
   begin
     if args(0) <> 1 then
       error("There must be exactly one argument")
     end_if;
     if x > n then
       new(dom, hold(many))
     else
       new(dom, x)
     end_if
   end_proc;

   print := proc(x) begin op(x, 1) end_proc;

   _plus := proc() local argv;
   begin
     argv:= map([args()], op, 1);
     if has(argv, hold(many)) then
        new(dom, hold(many))
     else
        dom(_plus(op(argv)))
     end_if
   end_proc;
   // initialization part
   begin
     if args(0) <> 1 then
       error("Wrong number of arguments")
     end_if;
   end:
```

Now, `CountingUpTo` is a domain constructor:

```
>> type(CountingUpTo)

                         DomainConstructor
```

We have defined the domain constructor `CountingUpTo`, but we have not created a domain yet. This is done by calling the constructor:

```
>> CountingUpToNine := CountingUpTo(9);
   CountingUpToTen := CountingUpTo(10)

                          CountingUpTo(9)

                         CountingUpTo(10)
```

We are now able to create, output, and manipulate domain elements as in the previous example:

```
>> x := CountingUpToNine(3): y := CountingUpToNine(7):
   x, x + x, y, x + y, y + y

                        3, 6, 7, many, many

>> x := CountingUpToTen(3): y := CountingUpToTen(7):
   x, x + x, y, x + y, y + y

                         3, 6, 7, 10, many

>> delete CountingUpToNine, CountingUpToTen, CountingUpTo, x, y:
```

No domain constructor with the same name may be used again during the same session.

Example 3. Suppose that your domain does not really depend on a parameter, but that you need some of the other features of domain constructors. Then you may define a domain constructor `dc`, say, that is called without parameters. From such a domain constructor, you can construct exactly one domain `dc()`. Instead of defining the constructor via `domain dc() ... end` first and then using `d := dc()` to construct the domain `d`, say, you may directly enter `domain d ... end`, thereby saving some work.

Continuing the previous examples, suppose that we want to count up to three, knowing that we never want to count farther. However, we want to declare our domain to be an Abelian semigroup with a canonical representation of the elements. This is not possible with a construction of the domain using `newDomain` as in example 1: we have to use the keyword `domain`. You will notice at once that the following source code is almost identical to the one in the previous example — we just removed the dependence on the parameter `n`.

```
>> domain CountingUpToThree

   inherits Dom::BaseDomain;
   category Cat::AbelianSemiGroup;
   axiom Ax::canonicalRep;

   new := proc(x : Type::PosInt)
   begin
     if args(0) <> 1 then
       error("There must be exactly one argument")
     end_if;
     if x > 3 then
       new(dom, hold(many))
     else
       new(dom, x)
     end_if
   end_proc;

   print := proc(x) begin op(x, 1) end_proc;

   _plus := proc() local argv;
   begin
     argv:= map([args()], op, 1);
     if has(argv, hold(many)) then
        new(dom, hold(many))
     else
        dom(_plus(op(argv)))
     end_if
   end_proc;

   end:
```

Now, `CountingUpToThree` is a domain and not a domain contructor:

```
>> type(CountingUpToThree)

                                      DOM_DOMAIN
```

You may use this domain in the same way as `CountingUpTo(3)` in example 2.

Background:

- Only one domain with a given key may exist. If it is stored in two variables `S` and `T`, say, assigning or deleting a slot `slot(S, a)` implicitly also changes `slot(T, a)` (reference effect). This also holds if `a = "key"`.

You get no warning even if `T` is protected.

`DOM_PROC_ENV` – the data type of procedure environments

A procedure environment represents a procedure that is currently being executed: formal parameters and local variables have values.

Procedure environments do rarely become visible, and you do not need to manipulate them directly. They serve for only one purpose: if a procedure is generated inside another procedure, variable names in the body of the inner procedure that are not declared local there refer to the outer procedure, provided they are declared local in the outer procedure. (see the Programming Manual for more information on MuPAD 's scoping rules .) Consequently, the inner procedure must contain information on the current values of local variables of the outer procedure. Hence, the status of of the outer procedure is encoded into an object of type `DOM_PROC_ENV`, and that object is stored in the returned procedure as its twelfth operand.

Construction:

You never need to generate objects of this type.

Important Operations:

- There are no operations available.

Result of Evaluation: Evaluating an object of the domain type `DOM_PROC_ENV` returns itself.

Function Call: Calling a procedure environment as a function gives the procedure environment itself, regardless of the arguments. The arguments are *not* evaluated.

Operands: The number of operands of a procedure environment depends on the number of local and saved variables of the outer procedure. Details about the operands remain undocumented.

Example 1. The only occasion on which you should come across a procedure environment is the following: an outer procedure returns an inner procedure depending on formal parameters or local variables of the outer procedure:

```
>> outer :=
   proc(x)
   option escape;
   begin
     /* inner procedure to return : */
     y -> x + y
   end_proc:
   add5 := outer(5)

                                        y -> x + y
```

In spite of the (slightly confusing) output, `x` has a special meaning here: it points to the parameter `x` of `outer`. That parameter currently has the value `5` and won't be changed any more. To be able to access that value, the particular instance of `outer` in the status of being executed has to be stored in `add5`:

```
>> op(add5, 12)

                                   DOM_PROC_ENV(5667160)
```

Background:

- The integers appearing in the output of objects of type `DOM_PROC_ENV` have no mathematical meaning; they denote positions in memory.

`Factored` – the domain of objects kept in factored form

`Factored` is the domain of objects kept in factored form, such as prime factorization of integers, square-free factorization of polynomials, or the factorization of polynomials in irreducible factors.

Creating Elements:

- `Factored(list<, type><, ring>)`
- `Factored(f<, type><, ring>)`

Parameters:

`list` — a list of odd length
`f` — an arithmetical expression
`type` — a string (default: `"unknown"`)
`ring` — a domain of category `Cat::IntegralDomain` (default: `Dom::ExpressionField()`)

Details:

- The argument `list` must be a list of odd length and of the form `[u, f1, e1, f2, e2, ..., fr, er]`, where the entries u and f_i are elements of the domain `ring`, or can be converted into such elements. The e_i must be integers. Here, i ranges from 1 to r.

  See section "Operands" below for the meaning of the entries of that list.

  An error message is reported, if one of the list entries is of wrong type.

- An arithmetical expression `f` given as the first argument is the same as giving the list `[ring::one, f, 1]`.

  See section "Operands" below for the meaning of the entries of that list.

  `f` must be an element of the domain `ring`, or must be convertible into such an element, otherwise an error message would be given.

- The argument `type` indicates what is known about the factorization. Currently, the following types are known:

  - `"unknown"` – nothing is known about the factorization.
  - `"irreducible"` – the f_i are irreducible over the domain `ring`.
  - `"squarefree"` – the f_i are square-free over the domain `ring`.

  If this argument is missing, then the type of the created factored object is set to `"unknown"`.

  The type of factorization is known to any element of `Factored`. Use the methods `"getType"` and `"setType"` (see below) to read and set the type of factorization of a given factored object.

- The argument `ring` is the ring of factorization. It must be an integral domain, i.e., a domain of category `Cat::IntegralDomain`.

  If this argument is missing, then the domain `Dom::ExpressionField()` is used.

  The ring of factorization is known to any element of `Factored`. Use the methods `"getRing"` and `"setRing"` (see below) to read and set the ring of factorization of a given factored object.

- You can use the index operator `[ ]` to extract the operands of an element `f` of the domain `Factored`, i.e., `f[1] = u, f[2] = f1, f[3] = e1, ...`.

  For example, to extract all factors f_i, enter `f[2*i] $ i = 1..nops(f) div 2`. To extract all exponents e_i, enter `f[2*i + 1] $ i = 1..nops(f) div 2`.

  You can also use the methods `"factors"` and `"exponents"` (see below) to access the operands, i.e., the call `Factored::factors(f)` returns a list of the factors f_i, and `Factored::exponents(g)` returns a list of the exponents e_i ($1 \leq i \leq r$).

- The system functions `ifactor`, `factor` and `polylib::sqrfree` are the main application of this domain, they return their result in form of such factored objects (see their help pages for information about the type and ring of factorization).

  There may be no need to explicitly create factored objects, but to work with the results of the mentioned system functions.

- Note that an element of `Factored` is printed like an expression and behaves like that. As an example, the result of `f := factor(x^2 + 2*x + 1)` is an element of `Factored` and printed as `(x + 1)^2`. The call `type(f)` returns `"_power"` as the expression type of `f`.

- For an element `f` of `Factored`, the call `Factored::convert(f, DOM_LIST)` gives a list of all operands of `f`.

Operands: An element f of `Factored` consists of the $r+1$ operands $u, f_1, e_1, f_2, e_2, \ldots, f_r, e_r$, such that $f = u \cdot f_1^{e_1} \cdot f_2^{e_2} \cdot \ldots \cdot f_r^{e_r}$.

The first operand u and the factors f_i are elements of the domain `ring`. The exponents e_i are integers.

Important Operations:

- You can apply (almost) each function to factored objects, functions that mainly expect arithmetical expressions as their input.

  For example, one may add or multiply those objects, or apply functions such as `expand` and `diff` to them. But the result of such an operation then is usually not any longer of the domain `Factored`, as the factored form could be lost due to the operation (see examples below).

- Call `expr(f)` to convert the factored object `f` into an arithmetical expression (as an element of a kernel domain).

- The call `coerce(f, DOM_LIST)` returns a list of operands of the factored object `f` (see method `"convert_to"` below).

Result of Evaluation: Evaluating an object of the domain type `Factored` returns itself.

Function Call: Calling a factored object as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Mathematical Methods

Method `_mult`: multiply factored objects

`_mult(`*`Factored`*` f, `*`any`*` g, ...)`

- Computes the product $f \cdot g \cdot \ldots$.
- Suppose that `g` is an element of the domain `ring` (or can be converted into such an element).

  If `g` is a unit of `ring` or a factor of `f`, then the result is a factored object of the same factorization type as `f`. Otherwise, the result is an element of `Factored` with the factorization type `"unknown"`.
- If both `f` and `g` are factored objects with factorization type `"irreducible"`, then the result is again a factored object of this type, i.e., the result is still in factored form.
- Otherwise, the factored form of `f` is lost, and the result of this method is an element of `ring`.
- This method overloads the function `_mult` for factored objects, i.e., one may use it in the form `f*g*...`, or in functional notation: `_mult(f, g, ...)`.

Method `_power`: raise a factored object to a certain power

`_power(`*`Factored`*` f, `*`any`*` n)`

- Computes f^n.
- If `n` is a positive integer and `f` a factored object with factorization type `"irreducible"` or `"squarefree"`, then the result is still a factored object of this type.
- Otherwise, the factored form of `f` is lost, and the result of this method is an element of `ring`.
- This method overloads the function `_power` for factored objects, i.e., one may use it in the form `f^n`, or in functional notation: `_power(f, n)`.

Method `expand`: expand a factored object

`expand(`*`Factored`*` f)`

- Returns `f` in expanded form. The result of this method is an element of `ring`.

Method `exponents`: get the exponents of a factored object

`exponents(`*`Factored`*` f)`

- Returns a list of the exponents e_i $(1 \leq i \leq r)$ of `f`.

Method `factor`: factorize a factored object

`factor(`*`Factored`*` f)`

- Factorizes `f` into irreducible factors.
- If `f` already is of the factorization type `"irreducible"`, then this method just return `f`.

  Otherwise, this method converts `f` into an element of the domain `ring` and calls the method `"factor"` of `ring`.

  This method returns a factored object of the domain `Factored` with factorization type `"irreducible"`, if the factorization of `f` can be computed (otherwise, `FAIL` is returned).
- This method overloads the function `factor` for factored objects, i.e., one may use it in the form `factor(f)`.

Method `factors`: get the factors of a factored object

`factors(`*`Factored`*` f)`

- Returns a list of the factors f_i $(1 \leq i \leq r)$ of `f`.

Method `irreducible`: test if a factored object is irreducible

`irreducible(`*`Factored`*` f)`

- Returns `TRUE`, if `f` is irreducible over the integral domain `ring`, otherwise `FALSE`.
- The test on irreducible is trivial, if `f` has the factorization type `"irreducible"`.

  Otherwise, this method converts `f` into an element of `ring` and calls the method `"irreducible"` of `ring`. The value `FAIL` is returned, if the domain `ring` cannot test if `f` is irreducible.

Method `iszero`: test on zero for factored objects

`iszero(`*`Factored`*` f)`

- Returns `TRUE` if `f` is zero, and `FALSE` otherwise.
- This method overloads the function `iszero` for factored objects, i.e., one may use it in the form `iszero(f)`.

Method `sqrfree`: compute a square-free factorization of a factored object

`sqrfree(`*`Factored`*` f)`

- Factorizes `f` into square-free factors.
- If `f` already is of the factorization type `"squarefree"`, then this method just return `f`.

  Otherwise, this method converts `f` into an element of the domain `ring` and calls the method `"squarefree"` of `ring`.

  This method returns a factored object of the domain `Factored` with factorization type `"squarefree"`, if the square-free factorization of `f` can be computed (otherwise, `FAIL` is returned).
- This method overloads the function `polylib::sqrfree` for factored objects, i.e., one may use it in the form `polylib::sqrfree(f)`.

Access Methods

Method `_index`: extract an operand of a factored object

`_index(`*`Factored`*` f, `*`positive integer`*` i)`

- Returns the `i`-th operand of `f` (see above for a description of the operands of such elements).
- Responds with an error message, if `i` is greater than the number of operands of `f`.
- This method overloads the index operator `[ ]` for factored objects, i.e., one may use it in the form `f[i]`.

Method `getRing`: get the ring of factorization

`getRing(`*`Factored`*` f)`

- Returns the domain `ring` of `f`.

Method `getType`: get the type of factorization

`getType(`*`Factored`*` f)`

- Returns the factorization type `type` of `f`.

Method `has`: existence of an object in a factored object

`has(`*`Factored`*` f, `*`any`*` x, ...)`

- Test whether an operand of `f` contains `x`. See the system function `has` for a detailed description of the parameters.
- This method overloads the function `has` for factored objects, i.e., one may use it in the form `has(f, x, ...)`.

Method `map`: map a function to the operands of factored objects

map(*Factored* f, *function* func, ...)

- This method first converts `f` into the unevaluated expression `u*f1^e1*f2^e2*...*fr^er` where `u, f1, e1, ...` are the operands of `f`. Then the function `func` is mapped to that expression.

  Note that the result of this method is not longer an object of `Factored`!
- See the system function `map` for details.
- This method overloads the function `map` for factored objects, i.e., one may use it in the form map(f, *function* func, ...).

Method `nops`: the number of operands of a factored object

nops(*Factored* f)

- Returns the number of operands of `f` (see above for a description of the operands of such elements).
- This method overloads the function `nops` for factored objects, i.e., one may use it in the form `nops(f)`.

Method `op`: extract an operand of a factored object

op(*Factored* f, *positive integer* i)

- Returns the `i`-th operand of `f` (see above for a description of the operands of such elements).
- Returns `FAIL`, if `i` is greater than the number of operands of `f`.
- This method overloads the function `op` for factored objects, i.e., one may use it in the form `op(f, i)`.

Method `select`: select operands of a factored object

select(*Factored* f, *function* func, ...)

- Select all operands of `f` with respect to the decision function `func`. See the system function `select` for a detailed description of the parameters.
- This method overloads the function `select` for factored objects, i.e., one may use it in the form `select(f, func, ...)`.

Method `set_index`: set an operand of a factored object

`set_index(`*`Factored`*` f, `*`positive integer`*` i, `*`any`*` x)`

- Sets the `i`-th operand of `f` to the value of `x` (see above for a description of the operands of such elements).

  The factorization type of `f` is set to `"unknown"`.
- Responds with an error message, if `i` is greater than the number of operands of `f`.

  Make sure that `x` either is an element of the domain `ring`, or an integer.
- This method overloads the index operator `[ ]` for factored objects, i.e., one may use it in the form `f[i]`.

Method `setRing`: set the ring of factorization

`setRing(`*`Factored`*` f, `*`domain`*` ring)`

- Sets the factorization ring of `f` to the domain `ring`.
- Use this method with caution! Make sure that the factorization of `f` is still valid over the new ring, and that the operands of `f` have the correct domain type.

  `ring` must be a domain of category `Cat::IntegralDomain`, which is not checked by this method.

Method `setType`: set the type of factorization

`setType(`*`Factored`*` f, `*`string`*` type)`

- Sets the factorization type of `f` to `type`.
- Use this method with caution! Make sure that the factorization type corresponds with the factorization of `f`.

Method `subs`: substitute subexpressions in the operands of a factored object

`subs(`*`Factored`*` f, `*`equation`*` x = a, ...)`

- Substitute subexpressions in the operands of `f`. See the system function `subs` for a detailed description of the parameters.
- This method overloads the function `subs` for factored objects, i.e., one may use it in the form `subs(f, `*`equation`*` x = a, ...)`.

Method `subsop`: substitute operands of a factored object

`subsop(`*`Factored`*` f, `*`equation`*` i = a, ...)`

- Substitute the `i`-th operand of `f` by `a`. See the system function `subsop` for a detailed description of the parameters.
- This method overloads the function `subsop` for factored objects, i.e., one may use it in the form `subsop(f, `*`equation`*` i = a, ...)`.

Method `type`: expression type of factored objects

`type(`*`Factored`*` f)`

- This method converts `f` into the unevaluated expression `u*f1^e1*f2^e2*...*fr^er` and returns its expression type that is either `"_power"`, `"_mult"`, or the type of an element of the domain `ring`.

Conversion Methods

Method `convert`: convert an object into a factored object

`convert(`*`any`*` x)`

- This method tries to convert `x` into an element of the domain type `Factored`.
- If the conversion fails, then `FAIL` is returned.
- `x` may either be a list of the form `[u, f1, e1, ..., fr, er]` of odd length (where `u, f1, ..., fr` are of the domain type `ring`, or can be converted into such elements, and `e1, ..., er` are integers), or an element that can be converted into the domain `ring`. The latter case corresponds to the list `[ring::one,x,1]`.

Method `convert_to`: convert factored objects into other domains

`convert_to(`*`Factored`*` f, `*`any`*` T)`

- This method tries to convert the factored object `f` into an element of domain type `T`, or, if `T` is not a domain, to the domain type of `T`.
- If the conversion fails, then `FAIL` is returned.
- If `T` is the domain `DOM_LIST`, then the list of operands of `f` is returned.

  If `T` is the domain `DOM_EXPR`, then the unevaluated expression `u*f1^e1*f2^e2*...*fr^e` is returned, where `u, f1, e1, ...` are the operands of `f`.

  Otherwise, the method `"convert"` of the domain `T` is called to convert `f` into an element of the domain `T` (which could return `FAIL`).
- Use the function `expr` to convert `f` into an object of a kernel domain (see below).

Method `create`: create simple and fast a factored objects

`create(list list)`

- This method creates a new factored object in the usual way, assuming, that `list` have the correct form and type of elements (see the description of the operands of a factored object).

`create(ring x)`

- This method creates a new factored object with the operands `ring::one, x, 1`.

Method `expr`: convert a factored object into a kernel domain

`expr(Factored f)`

- This method converts `f` into an object of a kernel domain, applying the method `"expr"` of the domain `ring` to each factor of `f`.
- Note that the factored form of `f` may be lost due to this conversion.

Method `expr2text`: convert a factored object into a string

`expr2text(Factored f)`

- Converts `f` to a string.

Method `testtype`: type testing for factored objects

`testtype(Factored f, domain T)`

- Checks, if `f` can be converted into an element of the domain `T`, and returns `TRUE` or `FAIL`, respectively.

  This method uses the method `"convert"` (see above) into check, if a conversion is possible.
- This method is called from the system function `testtype`.

Method `TeX`: LaTeX formatting of a factored object

`TeX(Factored f)`

- Returns a LaTeX-formatted string for the factored object `f`.
- The method `"TeX"` of the domain `ring` is used to get the LaTeX-representation of the corresponding operands of `f`.
- This method is called from the system function `generate::TeX`.

Technical Methods

Method `_concat`: concatenate operands of factored objects

`_concat(`*`Factored`*` f, `*`Factored`*` g)`

- Returns a new factored object by appending the factors and exponents of `g` to the operands of `f`. The first operand of the new factored object is the first operand of `f` multiplied by the first operand of `g`.

  The factorization type of the new factored object is set to `"unknown"`.
- `f` and `g` must have the same factorization type and factorization ring, otherwise an error message is given.

Method `maprec`: allow recursive mapping for factored objects

`maprec(`*`Factored`*` f, `*`any`*` x, ...)`

- This method is called from the function `misc::maprec` and allows recursive mapping for factored objects. See the corresponding help page of this function for details.
- First `f` is converted into the unevaluated expression `u*f1^e1*f2^e2*...*fr^er`, where `u, f1, e1, ...` are the operands of `f`. Then the function `misc::maprec` is called with this expression as its first parameter.

  Note that the result of this method is not longer an object of `Factored`!

Method `print`: pretty-print routine for factored objects

`print(`*`Factored`*` f)`

- Pretty-print of the factored object `f`. This method is used from the system function `print` for printing factored objects to the screen.

Method `unapply`: create a procedure from a factored object

`unapply(`*`Factored`*` f<, `*`identifier`*` x>)`

- This method converts `f` into an expression `e` using the method `"expr"` (see above) and interprets this expression as a function in `x, ...`. It returns a procedure computing the expression `e`.
- This method overloads the function `fp::unapply` for factored objects, i.e., one may use it in the form `fp::unapply(f)`. See `fp::unapply` for details.

Example 1. The following computes the prime factorization of the integer 20:

```
>> f := ifactor(20)

                                    2
                                   2  5
```

The result is an element of the domain `Factored`:

```
>> domtype(f)

                                 Factored
```

which consists of the following five operands:

```
>> op(f)

                               1, 2, 2, 5, 1
```

They represent the integer 20 in the following form: $20 = 1 \cdot 2^2 \cdot 5^1$. The factors are prime numbers and can be extracted in two different ways:

```
>> Factored::factors(f), f[2*i] $ i = 1..nops(f) div 2

                               [2, 5], 2, 5
```

`ifactor` kept the information, that the factorization ring is the ring of integers (represented by the domain `Dom::Integer`), and that the factors of `f` are prime (and therefore irreducible, because $\mathbb{Z}$ is an integral domain):

```
>> Factored::getRing(f), Factored::getType(f)

                      Dom::Integer, "irreducible"
```

We can convert such an object into different forms, such as into a list of its operands:

```
>> Factored::convert_to(f, DOM_LIST)

                              [1, 2, 2, 5, 1]
```

or into an unevaluated expression, keeping the factored form:

```
>> Factored::convert_to(f, DOM_EXPR)

                                    2
                                   2  5
```

or back into an integer:

```
>> Factored::convert_to(f, Dom::Integer)

                                    20
```

You may also use the system function `convert` here, which has the same effect.

Example 2. We compute the factorization of the integers 108 and 512:

```
>> n1 := ifactor(108); n2 := ifactor(512)

                                              2  3
                                             2  3

                                               9
                                              2
```

The multiplication of these two integers gives the prime factorization of $55296 = 108 \cdot 512$:

```
>> n1*n2

                                             11  3
                                            2   3
```

Note that the most operations on such objects lead to an un-factored form, such as adding these two integers:

```
>> n1 + n2

                                              620
```

You may apply the function `ifactor` to the result, if you are interested in its prime factorization:

```
>> ifactor(%)

                                             2
                                            2  5 31
```

You an apply (almost) each function to factored objects, functions that mainly expect arithmetical expressions as their input. Note that, before the operation is applied, the factored object is converted into an arithmetical expression in un-factored form:

```
>> Re(n1)

                                              108
```

Example 3. The second system function which deals with elements of `Factored`, is `factor`, which computes all irreducible factors of a polynomial.

For example, if we define the following polynomial of $\mathbb{Z}_{101}$:

```
>> p := poly(x^12 + x + 1, [x], Dom::IntegerMod(101)):
```

and compute its factorization into irreducible factors, we get:

```
>> f := factor(p)

       2
 poly(x  + 73 x + 29, [x], Dom::IntegerMod(101)) poly(

     5       4       3       2
    x  + 62 x  + 64 x  + 63 x  + 58 x + 100, [x],

                                5       4       3        2
    Dom::IntegerMod(101)) poly(x  + 67 x  + 72 x  + 100 x  +

    33 x + 94, [x], Dom::IntegerMod(101))
```

If we multiply the factored object with an element that can be converted into an element of the ring of factorization, then we get a new factored object, which then is of the factorization type `"unknown"`:

```
>> x*f

       2
 poly(x  + 73 x + 29, [x], Dom::IntegerMod(101)) poly(

     5       4       3       2
    x  + 62 x  + 64 x  + 63 x  + 58 x + 100, [x],

                                5       4       3        2
    Dom::IntegerMod(101)) poly(x  + 67 x  + 72 x  + 100 x  +

    33 x + 94, [x], Dom::IntegerMod(101)) x

>> Factored::getType(%)

                                           "unknown"
```

You may use the function `expand` which returns the factored object in expanded form as an element of the factorization ring:

```
>> expand(f)

                 12
            poly(x   + x + 1, [x], Dom::IntegerMod(101))
```

Example 4. The third system function which return elements of `Factored` is `polylib::sqrfree`, which computes the square-free factorization of polynomials. For example:

```
>> f := polylib::sqrfree(x^2 + 2*x + 1)
```

```
                                               2
                                        (x + 1)
```

The factorization type, of course, is `"squarefree"`:

```
>> Factored::getType(f)

                                      "squarefree"
```

Changes:

- `Factored` is a new function.

`rectform` – the domain of expressions being splitted into real and imaginary part

`rectform` is the domain of arithmetical expressions being splitted into their real and imaginary part (if possible), i.e., an expression z of this domain is kept in the form $\Re(z) + i\Im(z)$, if possible.

Creating Elements:

- `rectform(z)`

Parameters:

`z` — an arithmetical expression

Details:

- See the help page of the system function `rectform` for a detailed description and examples about working with elements of the domain `rectform`.

Operands: An element z of `rectform` consists of three operands:

1. the real part of z,
2. the imaginary part of z,
3. the part of z, for that the real and imaginary part cannot be computed (possibly the integer 0, if there are not such subexpressions).

Related Domains: `DOM_COMPLEX`

Important Operations:

- You can apply (almost) each function to elements of `rectform`, functions which mainly expect arithmetical expressions as their input.

  For example, you may add or multiply those elements, or apply functions such as `expand` and `diff` to them. The result of such an operation, which is not explicitely overloaded by a method of `rectform` (see below), is an element of `rectform`.

  This "automatic overloading" works as follows: Each argument of the operation, which is an element of `rectform`, is converted to an expression using the method `"expr"` (see below). Then, the operation is applied and the result is re-converted to an element of `rectform`.
- Use the function `expr` to convert an element of `rectform` to an arithmetical expression (as an element of a kernel domain).
- The functions `Re` and `Im` return the real and imaginary part of elements of `rectform`.

Result of Evaluation: Evaluating an object of the domain type `rectform` returns itself.

Function Call: Calling an element of `rectform` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Mathematical Methods

Method `_mult`: multiply elements

`_mult(`*`rectform`* `z,` *`any`* `x, ...)`

- It tries to split the product $z \cdot x \cdot \ldots$ into its real and imaginary part and returns the result as an element of `rectform`.
- This method overloads the function `_mult` for elements of `rectform`, i.e., you may use it in the form `z*x*...`, or in functional notation: `_mult(z, x, ...)`.

Method `_plus`: add elements

`_plus(`*`rectform`* `z,` *`any`* `x, ...)`

- It tries to split the sum $z + x + \ldots$ into its real and imaginary part and returns the result as an element of `rectform`.

- This method overloads the function `_plus` for elements of `rectform`, i.e., you may use it in the form `z+x+ ...`, or in functional notation: `_plus(z, x, ...)`.

Method `_power`: raise an element to a certain power

`_power(`*`rectform`*` z, `*`any`*` x)`

- It tries to split z^x into its real and imaginary part and returns the result as an element of `rectform`.
- This method overloads the function `_power` for elements of `rectform`, i.e., you may use it in the form `z^x`, or in functional notation: `_power(z, x)`.

Method `conjugate`: the complex conjugate

`conjugate(`*`rectform`*` z)`

- Computes the complex conjugate of `z` and returns the result as an element of `rectform`.
- This method overloads the function `conjugate` for elements of `rectform`, i.e., you may use it in the form `conjugate(z)`.

Method `float`: floating point approximation

`float(`*`rectform`*` z)`

- Computes a floating point approximation of `z` by applying `float` to `z`. The result is of the domain type `DOM_FLOAT`.

  The precision of the approximation depends on the environment variable `DIGITS`.
- This method overloads the function `float` for elements of `rectform`, i.e., you may use it in the form `float(z)`.

Method `Re`: the real part

`Re(`*`rectform`*` z)`

- Returns the real part of `z`.
- This method overloads the function `Re` for elements of `rectform`, i.e., you may use it in the form `Re(z)`.

Method `Im`: the imaginary part

`Im(`*`rectform`*` z)`

- Returns the imaginary part of `z`.
- This method overloads the function `Im` for elements of `rectform`, i.e., you may use it in the form `Im(z)`.

Method `iszero`: test on zero

`iszero(`*`rectform`*` z)`

- Returns `TRUE` if `z` is zero, i.e, the three operands of `z` are zero.
- This method overloads the function `iszero` for elements of `rectform`, i.e., you may use it in the form `iszero(z)`.

Access Methods

Method `has`: existence of an object

`has(`*`rectform`*` z, `*`any`*` x, ...)`

- Test whether an operand of `z` contains `x`. See the system function `has` for a detailed description of the parameters.
- This method overloads the function `has` for elements of `rectform`, i.e., you may use it in the form `has(z, x, ...)`.

Method `nops`: the number of operands

`nops(`*`rectform`*` z)`

- Returns the number of operands of `z`, which is 3 for any element of `rectform`.
- This method overloads the function `nops` for elements of `rectform`, i.e., you may use it in the form `nops(z)`.

Method `op`: extract an operand

`op(`*`rectform`*` z, `*`positive integer`*` i)`

- Returns the `i`-th operand of `z` (see above for a description of the operands of such elements).
- Returns `FAIL`, if `i` is greater than 3 (i.e., the number of operands of `z`).
- This method overloads the function `op` for elements of `rectform`, i.e., you may use it in the form `op(z, i)`.

Method `subs`: substitute subexpressions

`subs(`*`rectform`*` z, `*`equation`*` x = a, ...)`

- Substitute subexpressions in the operands of `z`. See the system function `subs` for a detailed description of the parameters.
- This method overloads the function `subs` for elements of `rectform`, i.e., you may use it in the form `subs(z, x = a, ...)`.

Method `subsex`: substitute subexpressions (extended)

`subsex(`*`rectform`*` z, `*`equation`*` x = a, ...)`

- Substitute subexpressions in the operands of `z`. See the system function `subsex` for a detailed description of the parameters.
- This method overloads the function `subsex` for elements of `rectform`, i.e., you may use it in the form `subsex(z, x = a, ...)`.

Conversion Methods

Method `convert`: convert an object to this domain

`convert(`*`any`*` x)`

- This method converts `x` to an element of the domain type `rectform`, if `x` is an arithmetical expression (see `Type::Arithmetical`). Otherwise `FAIL` is returned.

Method `convert_to`: convert an element of this domain to other domains

`convert_to(`*`rectform`*` z, `*`any`*` T)`

- This method tries to convert `z` to an element of domain type `T`, or, if `T` is not a domain, to the domain type of `T`.
- It is implemented in the following way: First, `z` is converted to an arithmetical expression using the method `"expr"`. Then, the method `"convert"` of the domain `T` (or its domain type) is called to perform the conversion.
- Use the function `expr` to convert `z` to an object of a kernel domain.

Method `expr`: convert an element of this domain to a kernel domain

`expr(`*`rectform`*` z)`

- Converts `z` to an element of a kernel domain, i.e., the expression `o1 + I*o2 + o3` is returned, where `o1, o2` and `o3` are the operands of `z`.

Method `expr2text`: convert an element of this domain to a string

`expr2text(`*`rectform`*` z)`

- Converts `z` to a string.
- This method overloads the function `expr2text` for elements of `rectform`, i.e., you may use it in the form `expr2text(z)`.

Method `testtype`: type testing

`testtype(`*`rectform`*` z, `*`domain`*` T)`

- Checks, if `z` can be converted to an element of the domain `T`, and returns `TRUE` or `FALSE`, respectively.
- It is implemented in the following way: First, `z` is converted to an arithmetical expression using the method `"expr"`. After this, the function `testtype` is called and its result is returned.
- This method is called from the system function `testtype`.

Method `TeX`: LaTeX formatting

`TeX(`*`rectform`*` z)`

- Returns a LaTeX-formatted string for `z`.
- This method is called from the system function `generate::TeX`.

Technical Methods

Method `length`: length of an object

`length(`*`rectform`*` z)`

- Returns the length of `f`, which is the length of the expression `o1 + I*o2 + o3`, where `o1, o2` and `o3` are the operands of `z`.
- This method overloads the function `length` for elements of `rectform`, i.e., you may use it in the form `length(z)`.

Method `print`: pretty-print routine

`print(`*`rectform`*` z)`

- Returns the unevaluated expression `o1 + I*o2 + o3`, where `o1`, `o2` and `o3` are the operands of `z`.
- This method is used from the system function `print` for printing elements of `rectform` to the screen.

`Series::Puiseux` – **the domain of finite series expansions**

`Series::Puiseux` is a domain for finite series expansions. Elements of this domain represent initial segments of either Taylor, Laurent or Puiseux series expansions.

Details:

- The system function `series` is the main application of this domain. It tries to compute a Taylor, Laurent, or Puiseux series of a given arithmetical expression, and the result is returned as an element of `Series::Puiseux`.

  There may be no need for you to explicitely create elements of this domain, but to work with the results of `series`.

  Cf. the help page of `series` for a detailed description and examples of how to work with elements of the domain `Series::Puiseux`.

- Use the type expression `Type::Series` to determine for an element of this domain, which kind of series expansion it is.

- The coefficients are allowed to depend on the variable of the series expansion, for example, logarithmic terms in the series variable may appear as coefficients. Be aware that this is no Puiseux series in the mathematical sense. Cf. example 1.

Operands: A series of the domain type `Series::Puiseux` consists has operands:

1. the branching order b,
2. the valuation v,
3. the order of the error term e,
4. a list of coefficients $l_0, \ldots, l_{e-v-1}$,
5. the series variable x and the expansion point a in form of an equation $x = a$. The expansion point a may be infinity or -infinity as well.

The series looks as follows: $\sum_{i=0}^{e-v-1} l_i x^{(v+i)/b} + O(x^{e/b})$.

Related Domains: `Series::gseries`

Important Operations:

- `Series::Puiseux` implements basic arithmetic of series expansions. Use the ordinary arithmetical operators `+, -, *, /,^` and `@` for composition.
- The system functions `nthcoeff`, `coeff`, `nthterm`, `lterm`, `nthmonomial` and `lmonomial` as well as `ldegree` work on series expansions. Note that in contrast to polynomials, terms, coefficients and monomials we counted from the order term on. Cf. example 11.
- Use the function `expr` to convert a series expansion to an arithmetical expression (as an element of a kernel domain).

Result of Evaluation: Evaluating an object of the domain type `Series::Puiseux` returns itself.

Function Call: Calling an element of `Series::Puiseux` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Mathematical Methods

Method `diff`: differentiation

`diff(`*`Puiseux`*` s, `*`any`*` t)`

- Computes the formal derivative $\partial s/\partial t$ of a series expansion and returns the result as an element of `Series::Puiseux`.
- This method overloads the system function `diff`. Cf. example 2.

Method `_divide`: division

`_divide(`*`any`*` s, `*`Puiseux`*` t)`

- Tries to convert s into a series expansion via functions `new` and `series`. Then it computes the quotient s/t of series expansions and returns the result as an element of `Series::Puiseux`.
- This method overloads the system function `_divide` for series expansions, i.e., you may use it in the form `s/t`. Cf. example 3.

Method `_fconcat`: functional composition

`_fconcat(Puiseux s, Puiseux t, ...)`

- Computes the composition $s(t(\dots))$ of series expansions and returns the result as an element of `Series::Puiseux`.
- This method overloads the system function `_fconcat` for series expansions, i.e., you may use it in the form `s@t...`. Cf. example 4.

Method `int`: integration

`int(Puiseux s, any t)`

- Computes the formal integral $\int sdt$ of a series expansion and returns the result as an element of `Series::Puiseux`.
- This method overloads the system function `int`. Cf. example 2.

Method `_mult`: multiplication

`_mult(s, t)`

- At least one argument has to be a series expansion. Tries to convert the arguments into a series expansion via functions `new` and `series`. Then it computes the product $s \cdot t \cdot \dots$ of series expansions and returns the result as an element of `Series::Puiseux`.
- This method overloads the system function `_mult` for series expansions, i.e., you may use it in the form `s*t*...`. Cf. example 3.

Method `_plus`: addition

`_plus(Puiseux s, any t, ...)`

- At least one argument has to be a series expansion. Tries to convert the arguments into a series expansion via functions `new` and `series`. Then it computes the sum $s + t + \dots$ of series expansions and returns the result as an element of `Series::Puiseux`.
- This method overloads the system function `_plus` for series expansions, i.e., you may use it in the form `s+t+...`. Cf. example 3.

Method `_power`: exponentiation

`_power(Puiseux s, rational t)`

- Computes the power s^t of series expansions and returns the result as an element of `Series::Puiseux`.
- This method overloads the system function `_power` for series expansions, i.e., you may use it in the form `s^t`. Cf. example 5.

Method `revert`: functional inversion

`revert(`*`Puiseux`*` s)`

- Computes inverses of series expansions with respect to composition and returns the result as an element of `Series::Puiseux`.
- The expansion point of the inverse is the constant term of the series expansion.
- This method overloads the system function `revert`. Cf. example 4.

Method `Series::Puiseux::scalmult`: multiplication by a single monomial

`Series::Puiseux::scalmult(`*`Puiseux`*` s, `*`any`*` a, `*`rational`*` k )`

- Computes the product $s \cdot (ax^k)$ and returns the result as an element of `Series::Puiseux`. Cf. example 6.

Method `_subtract`: subtraction

`_subtract(`*`Puiseux`*` s, `*`any`*` t)`

- Computes the difference $s - t$ of series expansions and returns the result as an element of `Series::Puiseux`.
- This method overloads the system function `_subtract` for series expansions, i.e., you may use it in the form `s-t`. Cf. example 3.

Access Methods

Method `coeff`: extract coefficients

`coeff(`*`Puiseux`*` s <, `*`positive integer`*` n>)`

- Returns a sequence of all coefficients of `s`, or the `n`-th coefficient, counted from the lowest order term on, respectively.
- This method overloads the system function `coeff`. Cf. examples 11 and 7.

Method `Series::Puiseux::indet`: the series variable

`Series::Puiseux::indet(`*`Puiseux`*` s)`

- Returns the variable of `s`. Cf. example 10.

Method `iszero`: zero test

`iszero(Puiseux s)`

- Returns `TRUE` if `s` is zero, i.e., if the sum of the monomials of `s` is zero, that is if `s` only consists of an error term.
- This method overloads the system function `iszero`. Cf. example 8.

Method `lcoeff`: the leading coefficient (of lowest order)

`lcoeff(Puiseux s)`

- Returns the first (or leading) coefficient of `s` (see "Operands" for the definition of "coefficient"). This is the coefficient of the lowest order term.
- This method overloads the system function `lcoeff`. Cf. example 9.

Method `ldegree`: the leading degree (the valuation)

`ldegree(Puiseux s)`

- Returns the degree of the lowest order term of `s`, or the value `FAIL`, if `s` consists only of an error term.
- This method overloads the system function `ldegree`. Cf. example 10.

Method `lmonomial`: the leading monomial (of lowest order)

`lmonomial(Puiseux s)`

- Returns the first (or leading) monomial of `s`, i.e., the monomial of lowest order (see "Operands" for the definition of "monomial").
- This method overloads the system function `lmonomial`. Cf. example 9.

Method `lterm`: the leading term (of lowest order)

`lterm(Puiseux s)`

- Returns the first (or leading) term of `s`, i.e., the term of lowest order (see "Operands" for the definition of "term").
- This method overloads the system function `lterm`. Cf. example 9.

Method `nthcoeff`: extract coefficients

`nthcoeff(`*`Puiseux`*` s, `*`any`*` t)`

- Returns the `n`-th coefficient of `s`, counted from the lowest order monomial on (see “Operands” for the definition of “monomial”).
- This method overloads the system function `nthcoeff`. Cf. example 9

Method `nthmonomial`: extract monomials

`nthmonomial(`*`Puiseux`*` s, `*`positive integer`*` n)`

- Returns the `n`-th nonzero monomial of `s`, counted from the lowest order monomial on (see “Operands” for the definition of “monomial”).
- This method overloads the system function `nthmonomial`. Cf. example 9.

Method `nthterm`: extract terms

`nthterm(`*`Puiseux`*` s, `*`positive integer`*` n)`

- Returns the `n`-th nonzero term of `s`, counted from the lowest order term on (see “Operands” for the definition of “term”).
- This method overloads the system function `nthterm`. Cf. example 9.

Method `Series::Puiseux::order`: the order of the error term

`Series::Puiseux::order(`*`Puiseux`*` s)`

- Returns the order of the error term of `s`. Cf. example 10.

Method `Series::Puiseux::point`: the expansion point

`Series::Puiseux::point(`*`Puiseux`*` s)`

- Returns the expansion point of `s`. Cf. example 10.

Conversion Methods

Method `expr`: convert a series expansion into an element of a kernel domain

`expr(`*`Puiseux`*` s)`

- Returns the sum of the monomials of `s` without the order term.
- This method overloads the system function `expr`. Cf. example 12.

Method `float`: convert numeric parts of the coefficients into floats

```
float(Puiseux s)
```

- Returns a series expansion with floating-point coefficients.
- This method overloads the system function `float`. Cf. example 12.

Technical Methods

Method `Series::Puiseux::create`: create series expansion

```
Series::Puiseux::create(any bo, any v, any ord, list l,
identifier x, any x0)
```

- Creates a new series expansion with branch order `bo`, valuation `v` and order `ord` in `x` at point *x0*. The coefficients are in the list `l`.

Method `expand`: expand coefficients

```
expand(Puiseux s)
```

- Expands all coefficients of `s`.
- This method overloads the system function `expand`.

Method `Series::Puiseux::func_call`: evaluation at a point

```
Series::Puiseux::func_call(Puiseux s, any t)
```

- Evaluates the sum of the monomials of `s` without the error term at the point $x = t$, where x is the series variable.
- This method overloads the system function `evalp`. You may also use it in the form `s(t)`. Cf. example 13.

Method `has`: check whether an object occurs syntactically

```
has(Puiseux s, any t)
```

- Returns TRUE if `t` occurs syntactically in one of the coefficients, the series variable, or the point of expansion, and otherwise returns FALSE.
- This method overloads the system function `has`. Cf. example 12.

Method `map`: apply a function to all coefficients

`map(`*`Puiseux`*` s, `*`function`*` func, ...)`

- Applies the function `func` to all coefficients.
- This method overloads the system function `map`. Cf. example 14.

Method `Series::Puiseux::new`: create a new series expansion

`Series::Puiseux::new(`*`any`*` s, `*`identifier`*` x, `*`positive integer`*` n)`

- Computes a series expansion of `s` of order `n` with respect to the variable `x`. The point of expansion is the origin. Cf. example 15.

Method `print`: pretty-print routine

`print(`*`Puiseux`*` s)`

- This method is used by the system function `print` for printing elements of `Series::Puiseux` to the screen. Cf. example 16.

Method `subs`: replace subexpressions

`subs(`*`Puiseux`*` s, `*`equation`*` x = a, ...)`

- Replaces the subexpression `x` of `s` by `a`.
- This method overloads the system function `subs`. Cf. example 17.

Example 1.

```
>> f := series(psi(x), x = infinity);
   coeff(f, 0)

                 1      1       1       / 1  \
         ln(x) - --- - ----- + ------ + O| -- |
                 2 x       2        4    |  5 |
                       12 x    120 x     \ x  /

                             ln(x)
```

Example 2.

```
>> f := series(1 + 2*x^3, x);
   diff(f, x);
   int(f, x)

                               3      6
                        1 + 2 x  + O(x )

                             2      5
                          6 x  + O(x )

                              4
                             x       7
                         x + -- + O(x )
                             2

>> g := series(1 + 2*x^(3/2), x);
   diff(g, x);
   int(g, x)

                              3/2      6
                       1 + 2 x    + O(x )

                            1/2      5
                         3 x    + O(x )

                              5/2
                           4 x         7
                       x + ------ + O(x )
                             5

>> h := series(1 + 2*x, x = 2);
   diff(h, x);
   int(h, x)

                                            6
                   5 + (2 x - 4) + O((x - 2) )

                                      5
                         2 + O((x - 2) )

                                     2            7
             6 + (5 x - 10) + (x - 2)  + O((x - 2) )
```

Example 3.

```
>> f := series(1 + 2*x^3, x, 4);
   g := series(1 + 2*x^(3/2), x, 4);
   h := series(1 + 2*x, x = 2, 4)

                               3      4
                        1 + 2 x  + O(x )

                              3/2      4
                       1 + 2 x    + O(x )

                                            4
                   5 + (2 x - 4) + O((x - 2) )

>> f + g + h;
   _plus(f, g, h)

                              3/2      3      4
                 7 + 2 x + 2 x    + 2 x  + O(x )

                              3/2      3      4
                 7 + 2 x + 2 x    + 2 x  + O(x )

>> f - g;
   _subtract(f, g);
   g - h;
   _subtract(g, h)

                          3/2      3      4
                     - 2 x    + 2 x  + O(x )

                          3/2      3      4
                     - 2 x    + 2 x  + O(x )

                                  3/2      4
                   - 4 - 2 x + 2 x    + O(x )

                                  3/2      4
                   - 4 - 2 x + 2 x    + O(x )

>> f*g*h;
   _mult(f, g, h)

                         3/2      5/2       3      4
           5 + 2 x + 10 x    + 4 x    + 10 x  + O(x )

                         3/2      5/2       3      4
           5 + 2 x + 10 x    + 4 x    + 10 x  + O(x )
```

```
>> f/g;
   _divide(f, g);
   g/h;
   _divide(g, h)
                           3/2      3      4
                    1 - 2 x    + 6 x  + O(x )

                           3/2      3      4
                    1 - 2 x    + 6 x  + O(x )

                  3/2      2      5/2      3      7/2
         2 x   2 x      4 x    4 x      8 x    8 x         4
   1/5 - --- + ------ + ---- - ------ - ---- + ------ + O(x )
         25      5      125      25     625     125

                  3/2      2      5/2      3      7/2
         2 x   2 x      4 x    4 x      8 x    8 x         4
   1/5 - --- + ------ + ---- - ------ - ---- + ------ + O(x )
         25      5      125      25     625     125
```

Example 4.

```
>> f := series(1 + 2*x^3, x, 10);
   g := series(y^2, y, 10);
   f@g = _fconcat(f, g)
                               3      10
                        1 + 2 x  + O(x  )

                            2      10
                           y  + O(y  )

                     6      10           6      10
              1 + 2 y  + O(y  ) = 1 + 2 y  + O(y  )
>> f := series(1 + 2*x^(3/2), x, 10);
   g := series(y^2, y, 10);
   f@g = _fconcat(f, g)
                              3/2      10
                       1 + 2 x    + O(x  )

                            2      10
                           y  + O(y  )

                      3      9           3      9
               1 + 2 y  + O(y ) = 1 + 2 y  + O(y )
```

```
>> f := series(1 + 2*x^3, x = 2);
   g := series(y^2, y, 10);
   f@g = _fconcat(f, g)

                                 2            3            6
    17 + (24 x - 48) + 12 (x - 2)  + 2 (x - 2)  + O((x - 2) )

                            2      10
                           y  + O(y  )
 Error: invalid composition [Series::Puiseux::_fconcat]
```

We now consider the procedure `revert`. Let *f* be a series expansion where the constant term is zero.

```
>> f := series(5*x + 2*x^3, x);
   g := revert(f);
   f@g

                                3      6
                       5 x + 2 x  + O(x )

                           3       5
                    x   2 x    12 x       6
                    - - ---- + ----- + O(x )
                    5   625    78125

                                   6
                            x + O(x )
```

Otherwise the expansion point is the constant term.

```
>> f := series(1 + x + 2*x^3, x);
   g := revert(f);
   f@g

                                 3      6
                      1 + x + 2 x  + O(x )

                           3             5            6
        (x - 1) - 2 (x - 1)  + 12 (x - 1)  + O((x - 1) )

                                           6
                    1 + (x - 1) + O((x - 1) )
```

Example 5.

```
>> f := series(1 + 2*x^3, x);
   f^2 = _power(f, 2);
   f^(1/3) = _power(f, 1/3)

                               3      6
                        1 + 2 x  + O(x )

                      3      6            3      6
               1 + 4 x  + O(x ) = 1 + 4 x  + O(x )

                      3                  3
                   2 x       6        2 x       6
               1 + ---- + O(x ) = 1 + ---- + O(x )
                    3                  3

>> f := series(1 + 2*x^(3/2), x);
   f^2 = _power(f, 2);
   f^(1/3) = _power(f, 1/3)

                              3/2      6
                       1 + 2 x    + O(x )

             3/2      3      6            3/2      3      6
      1 + 4 x    + 4 x  + O(x ) = 1 + 4 x    + 4 x  + O(x )

        3/2      3       9/2
     2 x      4 x    40 x         6
 1 + ------ - ---- + ------- + O(x ) =
       3       9       81

           3/2      3       9/2
        2 x      4 x    40 x         6
    1 + ------ - ---- + ------- + O(x )
          3       9       81
```

Example 6.

```
>> f := series(1 + 2*x^3, x);
   Series::Puiseux::scalmult(f, 5, 3) = 5*x^3*f

                               3      6
                        1 + 2 x  + O(x )

              3       6      9       3       6      9
           5 x  + 10 x  + O(x ) = 5 x  + 10 x  + O(x )
```

```
>> f := series(1 + 2*x^3, x);
   Series::Puiseux::scalmult(f, 5) = 5*x^(0)*f

                               3      6
                        1 + 2 x  + O(x )

                      3      6            3      6
              5 + 10 x  + O(x ) = 5 + 10 x  + O(x )
```

Example 7.

```
>> f := series(5*x + 2*x^3, x);
   coeff(f) = (coeff(f, 1), coeff(f, 2), coeff(f, 3))

                                3      6
                       5 x + 2 x  + O(x )

                      (5, 0, 2) = (5, 0, 2)
```

Example 8.

```
>> f := series(5*x + 2*x^3, x);
   iszero(f);
   g := series(0, x);
   iszero(g)

                                3      6
                       5 x + 2 x  + O(x )

                              FALSE

                                 6
                              O(x )

                              TRUE
```

Example 9.

```
>> f := series(5*x + 2*x^3, x);
   nthcoeff(f, 1), nthcoeff(f, 2);
   lcoeff(f), lmonomial(f), lterm(f)
```

```
                                3      6
                       5 x + 2 x  + O(x )

                              5, 2

                            5, 5 x, x

>> nthmonomial(f, 1), nthmonomial(f, 2);
   nthterm(f, 1), nthterm(f, 2)

                                    3
                            5 x, 2 x

                                  3
                              x, x
```

Example 10.

```
>> f := series(5*x + 2*x^3, x);
   ldegree(f),
   Series::Puiseux::order(f),
   Series::Puiseux::indet(f),
   Series::Puiseux::point(f)

                                3      6
                       5 x + 2 x  + O(x )

                           1, 6, x, 0

>> g := series(5*x + 2*x^3, x = 3);
   ldegree(g),
   Series::Puiseux::order(g),
   Series::Puiseux::indet(g),
   Series::Puiseux::point(g)

                                 2            3            6
   69 + (59 x - 177) + 18 (x - 3)  + 2 (x - 3)  + O((x - 3) )

                           0, 6, x, 3
```

Example 11. Consider the series expansion $f := 5x + 2x^3 + O(x^6)$ of the polynomial $g := 5x + 2x^3$.

```
>> f := series(5*x + 2*x^3, x);  g := 5*x + 2*x^3
```

```
         3      6
5 x + 2 x  + O(x )

         3
5 x + 2 x
```

We have several access functions for series expansions overloading system functions for polynomials:

```
>> coeff(f, 1), coeff(g, 1);
   ldegree(f), ldegree(g)

5, 5

1, 1
```

Note, however, that the *n*-th term of a series expansion and the *n*-th term of a polynomial are different. For example, for polynomials, the leading monomial is the nonzero monomial with the highest degree, while for series expansions it is the nonzero monomial with the lowest degree.

```
>> lcoeff(f), lcoeff(g);
   nthmonomial(f, 1), nthmonomial(g, 1);
   lmonomial(f), lmonomial(g);
   lterm(f), lterm(g);
   nthterm(f, 1), nthterm(g, 1)

5, 2

        3
5 x, 2 x

        3
5 x, 2 x

    3
x, x

    3
x, x
```

Example 12.

```
>> f := series(1 + 2*x^3, x)

       3      6
1 + 2 x  + O(x )
```

```
>> expr(f);
   float(f)

                               3
                            2 x  + 1

                                 3      6
                      1.0 + 2.0 x  + O(x )

>> has(f, x);
   has(f, y)

                              TRUE

                              FALSE
```

Example 13.

```
>> f := series(1 + 2*x^3, x);
   Series::Puiseux::func_call(f, 3) = f(3);
   Series::Puiseux::func_call(f, y + 1) = f(y + 1)

                               3      6
                        1 + 2 x  + O(x )

                             55 = 55

                          3                3
                 2 (y + 1)  + 1 = 2 (y + 1)  + 1
```

Example 14.

```
>> f := series(1 + 2*x^3, x);
   map(f, sin)

                               3      6
                        1 + 2 x  + O(x )

                             3             6
                   sin(1) + x  sin(2) + O(x )
```

Example 15.

```
>> f := series(2*x^3, x);
   Series::Puiseux::new(2*x^3, x, 6);
   Series::Puiseux::create(1, 0, 6, [0, 0, 0, 2, 0, 0], x, 0)

                             3      6
                          2 x  + O(x )

                             3      9
                          2 x  + O(x )

                             3      6
                          2 x  + O(x )
```

```
>> f := series(sin(x), x) <> Series::Puiseux::new(sin(x), x, 6)

                  3    5
                 x    x        6                 6
             x - -- + --- + O(x ) <> sin(x) + O(x )
                 6    120
```

Example 16.

```
>> f := series(1 + 2*x^3, x):
   print(f):

                               3      6
                        1 + 2 x  + O(x )
```

Example 17.

```
>> f := series(1 + 2*x^3, x);
   subs(f, x = a);
   subs(f, x = y + 1)

                               3      6
                        1 + 2 x  + O(x )

                               3      6
                        1 + 2 a  + O(a )

                               2      3      6
                  3 + 6 y + 6 y  + 2 y  + O(y )
```

`Series::gseries` – the domain of generalized series expansions

`Series::gseries` is the domain of series expansions generalizing Taylor, Laurent and Puiseux expansions.

Creating Elements:

- `Series::gseries(f, x <, order>)`
- `Series::gseries(f, x = a <, order>)`
- `Series::gseries(f, x = a <, order> <, Right>)`
- `Series::gseries(f, x = a <, order> <, Left>)`

Parameters:

`f`	— an arithmetical expression
`x`	— the series variable: an identifier
`a`	— the expansion point: an arithmetical expression or ±`infinity`
`order`	— the truncation order: a nonnegative integer

Options:

`Left`	— compute a series expansion that is valid for real x smaller than a.
`Right`	— compute a series expansion that is valid for real x larger than a (the default case).

Return Value: an object of domain type `Series::gseries`, or the value `FAIL`.

Side Effects: The function is sensitive to the global variable `ORDER`, which determines the default number of terms of the expansion.

Details:

- The call `Series::gseries(f, x)` computes a series expansion at $x = 0+$.
- The system functions `series` and `asympt` are the main application of this domain. The latter function only returns elements of this domain, whereas `series` could return an element of `Series::gseries` in cases, where a Puiseux series expansion does not exist.

  There may be no need to explicitely create elements of this domain, but to work with the results of the mentioned system functions.

- See the help page of the system function `asympt` for a detailed description of the parameters and examples for working with elements of the domain `Series::gseries`.
- Note that elements of `Series::gseries` only represents *directional* (real) series expansions.

Operands: A series of the domain type `Series::gseries` consists of three operands:

1. A list of sublists $[c_i, f_i]$ of length 2. Each sublist represents a *monomial* $c_i \cdot f_i$ of the series expansion, where the c_i are the *coefficients* and f_i the *terms* of `s`.
2. The order term of the form $O\left(g(x)\right)$, possibly the integer 0, if the expansion is exact.
3. An arithmetical expression $e(x)$ depending of the series variable x such that $e(x) \to \infty$ for $x \to a+$.

Related Domains: `Series::Puiseux`

Important Operations:

- `Series::gseries` implements addition and multiplication of generalized series expansions. Use the ordinary arithmetical operators `+` and `*`.
- The system functions `coeff`, `nthterm`, `lterm`, `nthmonomial` and `lmonomial` as well as `ldegree` work on generalized series expansions.
- Use the function `expr` to convert a generalized series expansion into an arithmetical expression (as an element of a kernel domain).

Result of Evaluation: Evaluating an object of the domain type `Series::gseries` returns itself.

Function Call: Calling an element of `Series::gseries` as a function yields the object itself, regardless of the arguments. The arguments are *not* evaluated.

Mathematical Methods

Method `_mult`: multiply series expansions

```
_mult(gseries s, any t, ...)
```

- Computes the product $s \cdot t \cdot \ldots$ of series expansions and returns the result as an element of `Series::gseries`.

  If the product cannot be computed, then `FAIL` is returned.
- Each argument of this method, which is not of the domain type `Series::gseries`, is converted into such an element, i.e., a generalized series expansion is computed. If this fails, then `FAIL` is returned.
- This method overloads the function `_mult` for elements of `Series::gseries`, i.e., one may use it in the form `s*t*...`, or in functional notation: `_mult(s, t, ...)`.

Method `_plus`: add series expansions

```
_plus(gseries s, any t, ...)
```

- Computes the sum $s + t + \ldots$ of series expansions and return the result as an element of `Series::gseries`.

  If the sum cannot be computed, then `FAIL` is returned.
- Each argument of this method, which is not of the domain type `Series::gseries`, is converted into such an element, i.e., a generalized series expansion is computed. If this fails, then `FAIL` is returned.
- This method overloads the function `_plus` for elements of `Series::gseries`, i.e., one may use it in the form `s+t+ ...`, or in functional notation: `_plus(s, t, ...)`.

Method `_power`: the integer power of a series expansions

```
_power(gseries s, integer n)
```

- Computes the nth power of `s`, if `n` is a non-negative integer. Otherwise `FAIL` is returned.
- If `n` is not an integer, then `FAIL` is returned.
- This method overloads the function `_power` for elements of `Series::gseries`, i.e., one may use it in the form `s^n`, or in functional notation: `_power(s, n)`.

Method `coeff`: extract coefficients

```
coeff(gseries s<, positive integer n>)
```

- Returns a sequence of all coefficients of `s`, or the `n`th coefficient respectively.
- This method overloads the function `coeff` for elements of `Series::gseries`, i.e., one may use it in the form `coeff(s<, n>)`.

Method `iszero`: test on zero

`iszero(`*`gseries`*` s)`

- Returns `TRUE` if `s` is exact (i.e., the second operand is the integer 0) and equal to zero, otherwise `FALSE`.
- This method overloads the function `iszero` for elements of `Series::gseries`, i.e., one may use it in the form `iszero(s)`.

Method `lcoeff`: the leading coefficient

`lcoeff(`*`gseries`*` s)`

- Returns the first (or leading) coefficient of `s` (see "Operands" for the definition of the term "coefficient").
- This method overloads the function `lcoeff` for elements of `Series::gseries`, i.e., one may use it in the form `lcoeff(s)`.

Method `ldegree`: the leading degree

`ldegree(`*`gseries`*` s)`

- Returns the leading degree of `s`, or the value `FAIL`, if the leading degree cannot be determined.
- This method overloads the function `ldegree` for elements of `Series::gseries`, i.e., one may use it in the form `ldegree(s)`.

Method `lmonomial`: the leading monomial

`lmonomial(`*`gseries`*` s)`

- Returns the first (or leading) monomial of `s` (see "Operands" for the definition of the term "monomial").
- This method overloads the function `lmonomial` for elements of `Series::gseries`, i.e., one may use it in the form `lmonomial(s)`.

Method `lterm`: the leading term

`lterm(`*`gseries`*` s)`

- Returns the first (or leading) term of `s` (see "Operands" for the definition of "term").
- This method overloads the function `lterm` for elements of `Series::gseries`, i.e., one may use it in the form `lterm(s)`.

Method `nthmonomial`: extract monomials

`nthmonomial(`*`gseries`*` s, `*`positive integer`*` n)`

- Returns the `n`th monomial of `s` (see "Operands" for the definition of the term "monomial").
- This method overloads the function `nthmonomial` for elements of `Series::gseries`, i.e., one may use it in the form `nthmonomial(s, n)`.

Method `nthterm`: extract terms

`nthterm(`*`gseries`*` s, `*`positive integer`*` n)`

- Returns the `n`th term of `s` (see "Operands" for the definition of "term").
- This method overloads the function `nthterm` for elements of `Series::gseries`, i.e., one may use it in the form `nthterm(s, n)`.

Access Methods

Method `map`: map a function to coefficients

`map(`*`gseries`*` s, `*`function`*` func, ...)`

- Maps the function `func` to the coefficients of `s`.
- This method overloads the function `map` for elements of `Series::gseries`, i.e., one may use it in the form `map(s, func, ...)`.

Method `subs`: substitute subexpressions of monomials

`subs(`*`gseries`*` s, `*`equation`*` x = a, ...)`

- Substitute subexpressions of the monomials of the series `s`. See the system function `subs` for a detailed description of the parameters.
- This method overloads the function `subs` for elements of `Series::gseries`, i.e., one may use it in the form `subs(s, x = a, ...)`.

Method `subsex`: substitute subexpressions of monomials (extended)

`subsex(`*`gseries`*` s, `*`equation`*` x = a, ...)`

- Substitute subexpressions of the monomials of the series `s`. See the system function `subsex` for a detailed description of the parameters.
- This method overloads the function `subsex` for elements of `Series::gseries`, i.e., one may use it in the form `subsex(s, x = a, ...)`.

Conversion Methods

Method `convert`: convert an object into a generalized series expansion

`convert(`*`any`*` x)`

- If `x` is an element of the domain type `Series::Puiseux`, then `x` is converted into an element of `Series::gseries`.

  Otherwise, `FAIL` is returned.

Method `convert_to`: convert a generalized series expansion into other domains

`convert_to(`*`gseries`*` s, `*`any`*` T)`

- Tries to convert `s` into an element of domain type `T`, or, if `T` is not a domain, to the domain type of `T`.

  `FAIL` is returned, if a conversion cannot be performed, or is not supported.
- `T` might be the domain `DOM_POLY`, where the sum of monomials is considered as a polynomial in the indeterminates of the third operand of `s`.

  If `T` is the domain `DOM_EXPR`, then the conversion is the same as implemented by the method `"expr"` (see below).
- Use the function `expr` to convert `s` into an object of a kernel domain.

Method `create`: create simple and fast a generalized series expansion

`create(`*`list`*` list, `*`expression`*` orderTerm, `*`equation`*` x = a)`

- Creates a new element of the domain `Series::gseries`.

  `list` must be a list of sublists $[c_i, f_i]$ of length 2, the monomials $c_i \cdot f_i$ of the series expansion.

  `orderTerm` is the order term of the form $O\left(g(x)\right)$, or the integer 0 if the expansion is exact.
- This method should be used with caution, because no argument checking is performed. It is supposed to be used to *create*, not to compute elements of `Series::gseries`.

Method `expr`: convert a generalized series expansion into an element of a kernel domain

`expr(`*`gseries`*` s)`

- Returns the sum of monomials of `s` without the order term.

Technical Methods

Method `print`: pretty-print routine

`print(`*`gseries`*` s)`

- This method is used from the system function `print` for printing elements of `Series::gseries` to the screen.

Method `TeX`: LaTeX formatting

`TeX(`*`gseries`*` s)`

- Returns a LaTeX-formatted string for `s`.
- This method is called from the system function `generate::TeX`.