`Cat` — predefined categories

Table of contents

1 Introduction

In MuPAD an algebraic structure may be represented by a *domain*. Parametrized domains may be defined by *domain constructors*. Many domain constructors are defined in the library package `Dom`.

Domains which have a similar mathematical structure may be members of a *category*. A category adds a level of abstraction because it postulates conditions which must hold for a domain in order to become a valid member of the category. Operations may be defined for all members of a category based on the assumptions and basic operations of that category, as long as they make no assumptions about the representation of the elements of the domains that belong to the category. Categories may also depend on parameters and are created by *category constructors*.

Attributes of domains and categories are defined in terms of so-called *axioms*. Axioms state properties of domains or categories. They are defined by *axiom constructors*.

This paper describes the category constructors which are part of the `Cat` library package. The definition of new constructors is described in detail in the paper "Axioms, Categories and Domains" [2].

The categories defined so far in general follow the conventions of algebra. There are some properties of the categories which differ from the 'classical' non-constructive theory of algebra because these properties are not constructive or can not be constructed efficiently.

The category hierarchy of the `Cat` package is quite similar to (part of) the category hierarchy of AXIOM [3] (see [1] for a description of the basic categories of Scratchpad, the predecessor of AXIOM).

2 Category Constructors

For each category constructor only the entries defined directly by the constructor are described. Entries which are inherited from super-categories are not described.

Please note that most axioms of the categories are not stated explicitly. Only axioms which are not implied by the definition of a category are stated explicitly. The category of groups for example has no axiom stating that the multiplication is invertible because that is implied by the definition of a group.

Changes since Version 1.4

The definition and implementation of category constructors has been changed considerably:

- A new syntax has been introduced for category constructors.
- Constructor parameters and local variables are now bound lexically instead of substituted into the methods.
- The former spezial name `this` has been renamed to `dom`.

This changes are described in detail in the paper "Axioms, Categories and Domains" [2].

The category `Cat::SetCat` has been renamed to `Cat::BaseCategory`. Note that there is a new category `Cat::Set` which has nothing in common with the former `Cat::SetCat`.

Several new categories related to differential algebra have been introduced.

Changes since Version 1.2.2

Since MuPAD version 1.2.2 the following changes where made:

Most notably, all the constructors have been inserted into three library domains, in order to avoid global names and naming conflicts:

- All domain constructors and domains have been inserted into the new library domain `Dom`.
- The category constructors and categories have been inserted into the library domain `Cat`.
- The axioms have been inserted into the library domain `Ax`.

Thus the domain constructor for matrices now is called `Dom::Matrix` instead of simply `Matrix`, and the category of rings is called `Cat::Ring` instead of `Ring`.

The former global names may be exported from these library domains, with `export(Dom)` one gets all the former domain constructor and domain names for example.

The method names of the category `Cat::FactorialDomain` (formerly `FactorialDomain`) have been changed slightly, which involves the sub-categories and domains of this category.

Acknowledgements

Frank Postel[1] implemented the constructors related to matrices, i.e. `Cat::Matrix`, `Cat::SquareMatrix` and `Cat::VectorSpace` and wrote the documentation.

[1]Univ. of Paderborn, MuPAD group, e-mail: `frankp@uni-paderborn.de`

References

[1] J.H. Davenport and B.M. Trager. Scratchpad's View of Algebra I: Basic Commutative Algebra. *DISCO '90* (Springer LNCS 429, ed. A. Miola):40–54, 1990.

[2] K. Drescher. Axioms, Categories and Domains. *Automath Technical Report* No. 1, Univ. GH Paderborn 1995.

[3] R.D. Jenks and R.S. Sutor. *AXIOM, The Scientific Computation System*. Springer, 1992.

`Cat::BaseCategory` – the base category

`Cat::BaseCategory` is the "universal" category, any domain in the `Dom` package is of this category.

Generating the category:

- `Cat::BaseCategory`

Details:

- `Cat::BaseCategory` is the most general super-category of all categories defined by the `Cat` package. Any domain in the `Dom` package is of this category.
- The methods defined by `Cat::BaseCategory` are related to type conversion and equality testing, they are not related to an algebraic structure.

Basic Methods

Method `convert`: convert into this domain

```
convert(x)
```

- Must convert `x` into an element of this domain or return `FAIL` if that is not possible.

Method `convert_to`: convert to certain type

```
convert_to(dom x, type T)
```

- Must convert `x` into an element of type `T` or return `FAIL` if that is not possible. `T` may be domain or a type expression.

Method `equal`: test for equality

```
equal(dom x, dom y)
```

- This method must return `TRUE` if it can decide that `x` is equal to `y` in the mathematical sense imposed by this domain. It must return `FALSE` if it can decide that `x` is not equal to `y` mathematically. If the method can not decide the equality it must return `UNKNOWN`.
- Note that this method does *not* overload the function `_equal`, i.e. the = operator. The function `_equal` may not be overloaded.

Method `expr`: convert into expression

```
expr(dom x)
```

- Must convert `x` into an expression consisting of elements of kernel domains. The expression should canonically represent an element of this domain, if possible.

Conversion Methods

Method `coerce`: coerce into this domain

```
coerce(x)
```

- Tries to coerce `x` into an element of this domain. Must return `FAIL` if not successful.
- The implementation provided tries to convert `x` into an element of this domain by first calling `dom::convert(x)` and then, if this fails, `x::dom::convert_to(x, dom)`; it retuns `FAIL` if both methods fail.

Method `equiv`: test for equivalence

```
equiv(x, y)
```

- Tries to decide if the arguments are equal when regarded as elements of this domain. Returns `FAIL` if no decision was possible.
- The implementation provided tries to convert `x` and `y` into elements of this domain and then calls `dom::equal` with these elements. It returns `FAIL` if the conversion fails or the equality test returns `UNKNOWN`.

Method `new`: create element of this domain

```
new(x)
```

- Creates a new element of this domain.
- Given a domain `D`, an expression of the form `D(x,...)` results in a call of the form `D::new(x,...)`.
- The implementation provided here tries to convert `x` by calling `dom::convert(x)` and returns the result. It raises an error if `dom::convert` returns `FAIL`.

Method `print`: return expression to print an element

```
print(dom x)
```

- This method must return an expression which is printed instead of the original element `x`. It may be used to configure the printing of domain elements.
- Please do *not* print directly in this method by calling the function `print` for example!
- The implementation provided here is `dom::expr`.

Method `testtype`: test type of object

```
testtype(x, T)
```

- This method is called by the function `testtype`. It is used to test if `x` is of type `T`.
- This method must return `TRUE` if it can decide that `x` is of type `T`, `FALSE` if it can decide that `x` is not of type `T` and `FAIL` if it can not decide the test.
- This method is called in three different situations: Either if the argument `x` is of this domain, or if `T` is this domain, or if `T` is an element of this domain. Thus the following three situations can arise:

```
testtype(dom x, type T)
```

- In this case it must be tested if `x` may be regarded as an element of the type `T`, which may either be a domain or type expression. `dom::convert_to(x,T)` is called, if this is successful `TRUE` is returned and `FAIL` if not.

```
testtype(x, dom)
```

- In this case it must be tested if `x` may be regarded as an element of this domain. `dom::convert(x)` is called, if this is successful `TRUE` is returned and `FAIL` if not.

```
testtype(x, dom T)
```

- In this case `T` is regarded as a type expression. The implementation provided assumes that `T` represents the type consisting of the singleton element `T` and returns the result of the call `dom::equiv(x,T)`.

Technical Methods

Method `new_extelement`: create element of kernel or facade domain

```
new_extelement(x, ...)
```

- This method is defined only for domains with axiom `Ax::systemRep`, i.e. for kernel or facade domains. (Facade domains are domains which do not have elements of their own domain but operate on elements of kernel domains like `DOM_POLY`.)
- When an expression `new(D,x...)` is evaluated and `D` is a domain with method `"new_extelement"`, then `D::new_extelement(D,x...)` is evaluated and returned as result.
- Kernel or facade domains must define this method because otherwise the function `new` would return a "container" element of `D` rather than a "raw" element as intended.
- The implementation provided here returns the result of `D::new(x...)`. Thus a call of the form `new(D,x...)` yields the same result as a call of the form `D(x...)`.

Changes:

- Has been renamed. Used to be `Cat::SetCat`.
- The method `"printElem"` has been removed.

`Cat::AbelianGroup` – the category of abelian groups

`Cat::AbelianGroup` represents the category of abelian groups.

Generating the category:

- `Cat::AbelianGroup`

Categories:

`Cat::CancellationAbelianMonoid`

Details:

- An `Cat::AbelianGroup` is an abelian monoid with cancellation law where the operation + is invertible.

Basic Methods

Method `_negate`: returns opposite

`_negate(`*`dom`* `x)`

- Must return the opposite of `x`.

Mathematical Methods

Method `equal`: test for equality

`equal(`*`dom`* `x, `*`dom`* `y)`

- Returns `TRUE` iff `x` is equal to `y`. This implementation tests if `x` minus `y` is zero, using the method `"iszero"`.

Method `intmult`: returns integer multiple

`intmult(`*`dom`* `x, `*`DOM_INT`* `n)`

- Returns the integer multiple `n` times `x`. This method is implemented like "repeated squaring" using the domains method `"_plus"`.

Method `_subtract`: subtracts two elements

`_subtract(`*`dom`* `x, `*`dom`* `y)`

- Returns `x` minus `y` by adding `x` and the opposite of `y`.

Changes:

- No changes.

`Cat::AbelianMonoid` – the category of abelian monoids

`Cat::AbelianMonoid` represents an abelian monoid.

Generating the category:

- `Cat::AbelianMonoid`

Categories:

`Cat::AbelianSemiGroup`

Axioms

if `dom` has `Ax::canonicalRep` then
`Ax::normalRep`

Details:

- An `Cat::AbelianMonoid` is an abelian semi-group with a neutral element `dom::zero` according to the operation + (`_plus`).
- Use the axiom `Ax::normalRep` to state that zero is always represented in a unique way (i.e. canonically).
- If an abelian monoid has not the axion `Ax::normalRep` then `dom::zero` is only one possible representation of the neutral element. An abelian semi-group must at least have the method `"iszero"` to test for zero in such a case.

Basic Entries:

`zero` Must hold the neutral Element according to the operation +.

Mathematical Methods

Method `intmult`: returns integer multiple

```
intmult(dom x, Type::NonNegInt n)
```

- Returns `dom::zero` if `n` is 0 and the `n`-fold sum of `x` if `n` is positive. This method is implemented like "repeated squaring" using the domains method `"_plus"`.

Method `iszero`: tests if element is zero

```
iszero(dom x)
```

- Returns `TRUE` if `x` is equal to zero. This implementation uses the method `"equal"` to compare `x` with `dom::zero`.

Changes:

- No changes.

`Cat::AbelianSemiGroup` – the category of abelian semi-groups

`Cat::AbelianSemiGroup` represents the category of abelian semi-groups.

Generating the category:

- `Cat::AbelianSemiGroup`

Categories:

`Cat::BaseCategory`

Details:

- `Cat::AbelianSemiGroup` represents the category of abelian semi-groups where the operation is written as addition. Hence an `Cat::AbelianSemiGroup` is a set with an associative and commutative operation `+` (`_plus`).
- Note that non-abelian semi-groups with operation `*` have category `Cat::SemiGroup`.

Basic Methods

Method `_plus`: returns the sum of its arguments

`_plus(`*`dom`* `x, ...)`

- Must return the sum of its arguments.

Mathematical Methods

Method `intmult`: returns integer multiple

`intmult(`*`dom`* `x,` *`Type::PosInt`* `n)`

- Returns the `n`-fold sum of `x`. This method is implemented like "repeated squaring" using the domains method `"_plus"`.

Changes:

- No changes.

`Cat::Algebra` – the category of associative algebras

`Cat::Algebra(R)` represents the category of associative algebras over the commutative ring `R`.

Generating the category:

- `Cat::Algebra(R)`

Parameters:

`R` — A domain which is a commutative ring. The algebra will be an algebra over this ring.

Categories:

`Cat::Ring`, `Cat::Module(R)`

Details:

- An `Cat::Algebra(R)` is a module over a commutative ring `R` which also is a ring.

Changes:

- No changes.

`Cat::CancellationAbelianMonoid` – the category of abelian monoids with cancellation

`Cat::CancellationAbelianMonoid` represents the category of abelian monoids with cancellation.

Generating the category:

- `Cat::CancellationAbelianMonoid`

Categories:

`Cat::AbelianMonoid`

Details:

- A `Cat::CancellationAbelianMonoid` is an abelian monoid where the cancellation law holds according to the operation `+`, i.e. $a + b = a + c$ implies $b = c$.

Basic Methods

Method `_subtract`: subtracts two elements

```
_subtract(dom x, dom y)
```

- Must return `z` such that `x = y + z` or `FAIL` if `z` dosn't exist. The result is unique due to the cancellation law.

Mathematical Methods

Method `equal`: test for equality

`equal(`*`dom`* `x, `*`dom`* `y)`

- Returns `TRUE` if `x - y` exists and is equal to zero. Returns `FAIL` if `x - y` returns `FAIL`.
- The method `"iszero"` is used to test for zero.

Method `_negate`: negate element

`_negate(`*`dom`* `x)`

- Returns the opposite of `x` by computing 0 - `x` or `FAIL` if the subtraction fails.

Method `intmult`: returns integer multiple

`intmult(`*`dom`* `x, `*`DOM_INT`* `n)`

- Returns the `n`-fold sum of `x`. The integer `n` may also be negative. In this case the opposite of `x` is computed. If no opposite exists then `FAIL` is returned, otherwise the -`n`-fold sum of the opposite is returned.

Changes:

- No changes.

`Cat::CommutativeRing` – the category of commutative rings

`Cat::CommutativeRing` represents the category of commutative rings.

Generating the category:

- `Cat::CommutativeRing`

Categories:

`Cat::Ring`, `Cat::RightModule(dom)`

Details:

- A `Cat::CommutativeRing` is a ring with unit `dom::one` where the multiplication `*` (`_mult`) is commutative. It is also a right module over itself.
- This implementation additionally assumes that the elements are always constant with respect to differentiation and derivates. One must re-implement the methods `"diff"` and `"D"` if this assumption is false.

Mathematical Methods

Method `diff`: differentiates element

`diff(`*`dom`* `x <,` *`variable`* `v, ...>)`

- This implementation always returns 0.

Method `D`: returns derivative

`D(`*`Type::ListOf(Type::PosInt)`* `l,` *`dom`* `x)`

- This implementation always returns 0.

Changes:

- New methods: `"diff"` and `"D"`.

`Cat::DifferentialRing` – the category of ordinary differential rings

`Cat::DifferentialRing` represents the category of ordinary differential rings.

Generating the category:

- `Cat::DifferentialRing`

Categories:

`Cat::PartialDifferentialRing`

Details:

- A `Cat::DifferentialRing` is a commutative ring with a single derivation operator `D`.
- A derivation is a linear operator with product rule, i.e. $D(fg) = D(f)g + fD(g)$ holds for all f and g.

Basic Methods

Method `D`: returns derivative

`D(`*`dom`* `f)`

- Must return the derivative of `f`.

Method `diff`: differentiation with respect to a variable

`diff(`*`dom`* `f,` *`variable`* `x)`

- Must differentiate `f` with respect to the variable `x`.

Changes:

- No changes.

`Cat::EntireRing` – the category of entire rings

`Cat::EntireRing` represents the category of entire rings.

Generating the category:

- `Cat::EntireRing`

Categories:

`Cat::Ring`, `Cat::RightModule(dom)`

Axioms

`Ax::noZeroDivisors`

Details:

- An `Cat::EntireRing` is a ring with unit `"one"` which has no zero divisors: Given non-zero ring elements a and b the product a times b is never zero.

Changes:

- No changes.

`Cat::EuclideanDomain` – the category of euclidean domains

`Cat::EuclideanDomain` represents the category of euclidean domains.

Generating the category:

- `Cat::EuclideanDomain`

Categories:

`Cat::PrincipalIdealDomain`

Details:

- An `Cat::EuclideanDomain` is a principal ideal domain with an "Euclidean degree" function `"euclideanDegree"` and operations `"quo"` and `"rem"` computing the Euclidean quotient and Euclidean reminder.
- The Euclidean degree returns nonnegative integers such that for each non-zero x and y there exist s and r such that $x = ys + r$ and either the Euclidean degree of r is less than that of s or r is zero.
- In addition s is equal to `quo(x,y)` and r is equal to `rem(x,y)`.

Basic Methods

Method `euclideanDegree`: returns Euclidean degree

`euclideanDegree(`*`dom`* `x)`

- Must return the Euclidean degree of `x`.

Method `divide`: division with reminder

`divide(`*`dom`* `x, `*`dom`* `y)`

- Must return a list with two elements: first the Euclidean quotient and second the Euclidean reminder of `x` and `y`.

Mathematical Methods

Method `_divide`: exact division

`_divide(`*`dom`* `x, `*`dom`* `y)`

- Implements the exact division `_divide` in terms of `"divide"`: The division returns the Euclidean quotient of `x` and `y` provided that the Euclidean reminder is zero. It returns `FAIL` if the Euclidean reminder is not zero.

Method `gcd`: greatest common divisor

`gcd(`*`dom`* `x, ...)`

- Returns the greatest common divisor of its arguments computed by the Euclidean algorithm.

Method `gcdex`: extended greatest common divisor

`gcdex(`*`dom`* `x, `*`dom`* `y)`

- Returns a list `[g,s,t]` where g is the gcd of `x` and `y` and $g = xs + yt$ holds. The result is computed by the extended Euclidean algorithm.

Method `idealGenerator`: generator of finitely generated ideal

`idealGenerator(`*`dom`* `x, ...)`

- Returns the generator of the finitely generated ideal which is generated by the arguments. This is simply the gcd of the arguments.

Method `quo`: Euclidean quotient

`quo(`*`dom`* `x, `*`dom`* `y)`

- Returns the Euclidean quotient of `x` and `y`.
- The default implementation provided here uses the basic method `"divide"`.

Method `rem`: Euclidean reminder

```
rem(dom x, dom y)
```

- Returns the Euclidean reminder of `x` and `y`.
- The default implementation provided here uses the basic method `"divide"`.

Changes:

- No changes.

`Cat::FactorialDomain` – the category of factorial domains

`Cat::FactorialDomain` represents the category of factorial domains (i.e. unique factorisation domains).

Generating the category:

- `Cat::FactorialDomain`

Categories:

`Cat::GcdDomain`

Details:

- A `Cat::FactorialDomain` is an integral domain with gcd where an unique factorization can be computed.
- The factorization methods are named `"factor"` and `"sqrfree"` and must return elements of the domain `Factored` over this domain.

Basic Methods

Method `factor`: unique factorization

```
factor(dom x)
```

- Must return the unique factorization of `x` as an element of the domain `Factored` over this domain, such that the factors have the type `"irreducible"`.
- See `Factored` for details about the representation of the factorization.

Mathematical Methods

Method `irreducible`: tests if element is irreducible

`irreducible(`*`dom`* `x)`

- Returns `TRUE` if `x` is an irreducible element. The default implementation provided uses the method `"factor"` and therefore may be quite inefficient.

Method `sqrfree`: square-free factorization

`sqrfree(`*`dom`* `x)`

- Returns the square-free factorization of `x` as an element of the domain `Factored` over this domain, such that the factors have the type `"sqrfree"` or `"irreducible"`.
- See `Factored` for details about the representation of the factorization.
- The default implementation provided here uses the method `"factor"` and therefore may be very inefficient.

Changes:

- The methods `"factor"` and `"sqrfree"` must return elements of the domain `Factored` now.
- The method `"Factor"` was removed.

`Cat::Field` – the category of fields

`Cat::Field` represents the category of fields.

Generating the category:

- `Cat::Field`

Categories:

`Cat::EuclideanDomain`, `Cat::FactorialDomain`, `Cat::SkewField`

Axioms

`Ax::canonicalUnitNormal`, `Ax::closedUnitNormals`

Details:

- A `Cat::Field` is a factorial domain, an Euclidean domain and a skew field. As Euclidean domain it has a commutative multiplication `*` (`_mult`) and as skew field the multiplication is invertible.
- Many of the methods defined for factorial and Euclidean domains are trivial for a field.

Mathematical Methods

Method `associates`: test for associate elements

```
associates(dom x, dom y)
```

- Returns `TRUE` iff `x` and `y` are associate elements. For a field this is true iff both arguments are nonzero.

Method `_divide`: exact division

```
_divide(dom x, dom y)
```

- Returns `x * y^(-1)`.

Method `divide`: division with reminder

```
divide(dom x, dom y)
```

- Returns the list `[_divide(x,y), dom::zero]`.

Method `divides`: test if division is exact

```
divides(dom x, dom y)
```

- Always returns `TRUE`.

Method `euclideanDegree`: returns Euclidean degree

```
euclideanDegree(dom x)
```

- Returns `0` if `x` is zero and `1` otherwise.

Method `factor`: unique factorization

```
factor(dom x)
```

- Returns a trivial factorization, consisting of `x` to the power of 1 only. The factorization is returned as an object of the domain `Factored` and represents an irreducible factorization over this domain.

Method `gcd`: greatest common divisor

```
gcd(dom x, ...)
```

- Returns the gcd of the arguments: `dom::one` if at least one argument is nonzero and `dom::zero` otherwise.

Method `irreducible`: tests if element is irreducible

```
irreducible(dom x)
```

- Always returns `FALSE`.

Method `isUnit`: tests if element is an unit

```
isUnit(dom x)
```

- Returns `TRUE` iff `x` is nonzero.

Method `quo`: returns Euclidean quotient

```
quo(dom x, dom y)
```

- Returns `_divide(x, y)`.

Method `rem`: returns Euclidean reminder

```
rem(dom x, dom y)
```

- Always returns `dom::zero`.

Method `sqrfree`: square-free factorization

```
sqrfree(dom x)
```

- Returns a trivial factorization, consisting of `x` to the power of 1 only. The factorization is returned as an object of the domain `Factored` and represents an irreducible factorization over this domain.

Method `unitNormal`: unit normal form

```
unitNormal(dom x)
```

- Returns `dom::zero` if `x` is zero and `dom::one` otherwise.

Method `unitNormalRep`: unit normal representation

```
unitNormalRep(dom x)
```

- Returns the list `[dom::one, x^(-1), x]` if `x` is nonzero and `[dom::zero, dom::one, dom::one]` if `x` is zero.

Changes:

- No changes.

`Cat::FiniteCollection` – the category of finite collections

`Cat::FiniteCollection` represents the category of finite collections, i.e., the category of "universal" bags.

Generating the category:

- `Cat::FiniteCollection`

Categories:

```
Cat::BaseCategory
```

Details:

- A finite collection is a data structure where each element represents a finite bag of "things" of any type.
- The elements are numbered 1,...,`nops(c)`, where `nops(c)` is the number of elements in the bag.

Basic Methods

Method `_index`: returns element given its index

```
_index(dom x, Type::PosInt i)
```

- Must return the `i`-th element of `x`.

Method `map`: maps function on elements

```
map(dom x, function f <, a, ...>)
```

- Must replace each element `e` of `x` by `f(e,a...)` and return the result.

Method `nops`: returns number of elements

```
nops(dom x)
```

- Must return the number of elements of `x`.

Method `op`: returns certain elements

```
op(dom x)
```

- Must return a sequence of all elements of `x`.

```
op(dom x, Type::PosInt i)
```

- Must return the `i`-th element of `x` or `FAIL` if an element with the given index does not exist.
- Operand ranges or pathes need not be handled by this method because they are handled directly by `op`.

Method `set_index`: changes element with given index

```
set_index(dom x, Type::PosInt i, v)
```

- Must replace the `i`-th element of `x` by `v`.
- Overloads the function `_assign`. The result is assigned to `x`.

Method `subs`: substitute in elements

```
subs(dom x, e = f)
```

- In each element of `x` the expression `e` must be substituted by `f`.

Method `subsop`: substitute operands

```
subsop(dom x, Type::PosInt i = v)
```

- Must replace the `i`-th element of `x` by `v`.

Technical Methods

Method `mapCanFail`: maps function on elements

`mapCanFail(`*`dom`* `x, `*`function`* `f <, a, ...>)`

- Replaces each element `e` of `x` by `f(e,a...)`. If one of the results of the calls is `FAIL`, then `FAIL` is returned.

Method `testEach`: test each element with a predicate

`testEach(`*`dom`* `x, `*`function`* `f <, a, ...>)`

- For each element `e` of `x` the call `f(e,a...)` is evaluated. The calls must return boolean values. If one of the results is not `TRUE` then `FALSE` is returned, `TRUE` otherwise.

Method `testOne`: tests if element exists fulfilling a predicate

`testOne(`*`dom`* `x, `*`function`* `f <, a, ...>)`

- For each element `e` of `x` the call `f(e,a...)` is evaluated. The calls must return boolean values. If one of the results is `TRUE` then `TRUE` is returned, `FALSE` otherwise.

Changes:

- Has been renamed. Used to be `Cat::FiniteCollectionCat`.

`Cat::GcdDomain` – the category of integral domains with gcd

`Cat::GcdDomain` represents the category of integral domains with a gcd.

Generating the category:

- `Cat::GcdDomain`

Categories:

`Cat::IntegralDomain`

Details:

- A `Cat::GcdDomain` is an integral domain where the greatest common divisor of two elements can be computed by the method `"gcd"`.

Basic Methods

Method `gcd`: greatest common divisor

`gcd(`*`dom`* `x, ...)`

- Must return the greatest common divisor of its arguments.
- The method must satisfy the following conditions:
  1. `x` and `y` must divide `dom::gcd(x,y)`,
  2. if `z` divides both `x` and `y`, then `z` must divide `dom::gcd(x,y)`,
  3. if a domain has the axiom `Ax::canonicalUnitNormal` then `dom::gcd(x,y)` must be equal to `dom::unitNormal(dom::gcd(x,y))`.

  Remember that `x` divides `y` if `_divide(x,y)` does not return `FAIL`.

Mathematical Methods

Method `lcm`: least common multiple

`lcm(`*`dom`* `x, ...)`

- Returns the least common multiple of its arguments. The implementation provided here uses the method `"gcd"` to compute the result.

Changes:

- No changes.

`Cat::Group` – the category of groups

`Cat::Group` represents the category of groups.

Generating the category:

- `Cat::Group`

Categories:

`Cat::Monoid`

Details:

- A `Cat::Group` is a non-abelian monoid where the group operation * (`_mult`) is invertible.

Mathematical Methods

Method `_divide`: returns quotient

`_divide(`*`dom`* `x, `*`dom`* `y)`

- Returns the quotient `x/y` by computing `x*y^(-1)`.

Changes:

- No changes.

`Cat::HomogeneousFiniteCollection` – the category of homogeneous finite collections

`Cat::HomogeneousFiniteCollection(T)` represents the category of homogeneous finite collections (i.e. bags) of elements of the domain `T`.

Generating the category:

- `Cat::HomogeneousFiniteCollection(T)`

Parameters:

`T` — A domain which must be from the category `Cat::BaseCategory`. Only elements of this domain may be contained in the collection.

Categories:

`Cat::FiniteCollection`,

if `T` has `Cat::OrderedSet` then
`Cat::OrderedSet`

Details:

- A `Cat::HomogeneousFiniteCollection` is a finite collection where each element of the collection must be from the same domain `T`.

Entries:

`elemDom` The parameter domain `T`.

Mathematical Methods

Method `_less`: test if element is less

```
_less(dom x, dom y)
```

- This method is defined only if `T` is an ordered set.
- Returns `TRUE` if `x` is less than `y`.
- The collections `x` and `y` are ordered by the lexical ordering of their elements.

Changes:

- Has been renamed. Used to be `Cat::HomogeneousFiniteCollectionCat`.

`Cat::HomogeneousFiniteProduct` – the category of homogeneous finite products

`Cat::HomogeneousFiniteProduct(T)` represents the category of homogeneous finite products of elements of the domain `T`.

Generating the category:

- `Cat::HomogeneousFiniteProduct(T)`

Parameters:

`T` — A domain which must be from the category `Cat::BaseCategory`. This defines the domain of the products elements.

Categories:

if `T` is a `Cat::DifferentialRing` then
`Cat::DifferentialRing`

if `T` is a `Cat::PartialDifferentialRing` then
`Cat::PartialDifferentialRing`

if `T` is a `Cat::CommutativeRing` then
`Cat::CommutativeRing`

if `T` is a `Cat::SkewField` then
`Cat::SkewField`

if `T` is a `Cat::Ring` then
`Cat::Ring`

if T is a `Cat::Rng` then
`Cat::Rng`

if T is a `Cat::AbelianGroup` then
`Cat::AbelianGroup`

if T is a `Cat::CancellationAbelianMonoid` then
`Cat::CancellationAbelianMonoid`

if T is a `Cat::AbelianMonoid` then
`Cat::AbelianMonoid`

if T is a `Cat::AbelianSemiGroup` then
`Cat::AbelianSemiGroup`

if T is a `Cat::Group` then
`Cat::Group`

if T is a `Cat::Monoid` then
`Cat::Monoid`

if T is a `Cat::SemiGroup` then
`Cat::SemiGroup`

if T is a `Cat::CommutativeRing` then
`Cat::Algebra(T)`

if T is a `Cat::Ring` then
`Cat::LeftModule(T)`

if T is a `Cat::Ring` then
`Cat::RightModule(T)`

`Cat::HomogeneousFiniteCollection(T)`

Details:

- A `Cat::HomogeneousFiniteProduct(T)` is a homogeneous finite collection where each collection has the same number of elements of the domain `T`.
- The number of elements must be given by the entry `"card"`, which must be defined by domains of this category. It is not given as a category parameter simply because it is not needed. Thus no unnecessary instances of the category are created.
- One could principally implement all the algebraic operations here, but they will be slow if the methods `"_index"` and `"set_index"` are slow, which most often will be the case. So we avoid the work and let the domain implementors do it.

Basic Entries:

`card` Must hold the number of elements of a collection.

Entries:

`characteristic` Defined if `T` is a ring: In this case the characteristic of the product domain is the same as that of `T`.

Basic Methods

Method `zip`: combine elements

`zip(`*`dom`* `x, `*`dom`* `y, `*`function`* `f)`

- Must call `f(x_i, y_i)` for each pair `x_i`, `y_i` of elements from `x` and `y` and builds a new element of this domain from the results.

Method `zipCanFail`: combine elements, may fail

`zipCanFail(`*`dom`* `x, `*`dom`* `y, `*`function`* `f)`

- Must return the same as `zip(x, y, f)` with one difference: Must return `FAIL` if one of the results of `f` is `FAIL`.

Access Methods

Method `nops`: returns number of elements

`nops(`*`dom`* `x)`

- Returns the number of elements of `x`, which is simply the constant defined by the entry `"card"`.

Changes:

- Has been renamed. Used to be `Cat::HomogeneousFiniteProductCat`.

`Cat::IntegralDomain` – the category of integral domains

`Cat::IntegralDomain` represents the category of integral domains.

Generating the category:

- `Cat::IntegralDomain`

Categories:

`Cat::EntireRing`, `Cat::CommutativeRing`,
`Cat::Algebra(dom)`

Details:

- An `Cat::IntegralDomain` is a commutative and entire ring which has a "partial" division method `"_divide"`: If `b` divides `a` then `dom::_divide(a,b)` must return the quotient, otherwise `FAIL`. The result of the method `"_divide"` must be unique.
- Use the axiom `Ax::canonicalUnitNormal` to state in addition that there exists a canonical unit normal form for each element of the ring. If a ring has the axiom `Ax::canonicalUnitNormal` the method `"unitNormal"` must return the unique unit normal for a ring element. If the axiom is not valid the method may return any associate.
- Use the axiom `Ax::closedUnitNormals` in addition to state that the unit normals which are computed by the method `"unitNormal"` are closed under multiplication, i.e. that the product of two unit normals returns a unit normal.
- These two axioms are not implizitly valid for an `Cat::IntegralDomain` because there are integral domains for which one can't compute a canonical unit normal for each element.

Basic Methods

Method `_divide`: returns quotient

`_divide(`*`dom`* `x, `*`dom`* `y)`

- Must return the quotient `d` such that `x = d * y` or `FAIL` if such a quotient does not exist. The quotient is unique if it exists.
- The result must be unique:
  1. the product `y * dom::_divide(x,y)` must be equal to `x` provided that `y` is not zero and `y` divides `x`,
  2. if `x` is equal to `y * z` then `y` must divide `x`.
- It is an error if `y` is zero.

Method `isUnit`: tests if element is a unit

`isUnit(`*`dom`* `x)`

- Must return `TRUE` if `x` is an unit of the ring.

Method `unitNormal`: returns an associate

```
unitNormal(dom x)
```

- Must return an associate of `x`.
- If the ring has the axiom `Ax::canonicalUnitNormal` the method must return the unique unit normal of `x`.
- An implementation is provided if the ring has *not* the axiom `Ax::canonicalUnitNormal`. In this case simply `x` is returned.

Mathematical Methods

Method `associates`: tests if elements are associates

```
associates(dom x, dom y)
```

- Returns `TRUE` if `x` and `y` are associates. The implementation provided here uses the method `"divides"` to test if each argument divides the other.

Method `divides`: tests if elements divides another

```
divides(dom x, dom y)
```

- Returns `TRUE` if `x` divides `y`. The implementation uses the method `"_divide"` to test if `x` divides `y`.

Method `unitNormalRep`: returns the unit normal representation

```
unitNormalRep(dom x)
```

- Returns a list `[n, u, v]` where `n` is a unit normal form of `x`, `u` is a unit such that `n = u * x` and `v` is the inverse of `u`.
- If the ring has the axiom `Ax::canonicalUnitNormal` the method must return the unique unit normal of `x`. The default implementation uses the method `"unitNormal"` to compute the unit normal `n` in this case.
- If the ring dos not have the axiom `Ax::canonicalUnitNormal` the method simply returns `[x, dom::one, dom::one]`.

Changes:

- No changes.

`Cat::LeftModule` – the category of left R-modules

`Cat::LeftModule(R)` represents the category of left `R`-modules.

Generating the category:

- `Cat::LeftModule(R)`

Parameters:

`R` — A domain which must be from the category `Cat::Rng`.

Categories:

`Cat::AbelianGroup`

Details:

- A `Cat::LeftModule(R)` is an abelian group together with a rng `R` (a ring without unit) and a left multiplication `*` (`_mult`).
- The left multiplication is an operation taking an element of rng `R` and a module element and returning a module element.
- Given ring elements a, b and module elements x, y the following 3 distibutive laws must hold:

  1. $(ab)x = a(bx)$,
  2. $(a + b)x = ax + bx$,
  3. $a(x + y) = ax + ay$.

- Beware: The operation of a non-abelian semi-group is also written as `*` (`_mult`). The method `"_mult"` must handle the situation if a left module is also a non-abelian semi-group. In such a case it must both implement the group operation and the left multiplication by elements of the rng.

Basic Methods

Method `_mult`: left multiplication by a rng element

```
_mult(R r, dom x)
```

- Must return the left multiplication of `x` by the rng element `r`.

Changes:

- No changes.

`Cat::Matrix` – the category of matrices

`Cat::Matrix(R)` represents the category of matrices over the rng `R`.

Generating the category:

- `Cat::Matrix(R)`

Parameters:

`R` — A domain which must be from the category `Cat::Rng` (a ring without unit).

Categories:

`Cat::BaseCategory`

Details:

- A `Cat::Matrix(R)` is a matrix of arbitrary dimension over a component ring `R`.
- In the following description of the methods, we use the following notations for a matrix A from a domain of category `Cat::Matrix(R)`:

  nrows(A) denotes the number of rows and ncols(A) the number of columns of A.

  Further on, a *row index* is an integer ranges from 1 to nrows(A), and a *column index* is an integer ranges from 1 to ncols(A).

Entries:

`coeffRing` is set to `R`.

Basic Methods

Method `_index`: matrix indexing

```
_index(dom A, row index i, column index j)
```

- Must return the (i, j)-th component of the matrix `A`.

Method `matdim`: matrix dimension

```
matdim(dom A)
```

- Must return the number of rows and columns of the matrix `A` in form of a list of two positive integers.

Method `new`: matrix definition

```
new(positive integers m, n)
```

- Must return the $m \times n$ zero matrix.
- Of course, this method may implement further possibilites to create matrices (for example, see the method `"new"` of the domain constructor `Dom::Matrix`).

Method `set_index`: setting matrix components

```
set_index(dom A, row index i, column index j, R x)
```

- Must replace the (i, j)-th component of the matrix `A` by `x`.

Mathematical Methods

Method `_negate`: negates a matrix

```
_negate(dom A)
```

- Computes $-A$.

Method `_plus`: adds matrices

```
_plus(dom A1, dom A2, ..., dom An)
```

- Returns the sum $A_1 + A_2 + \ldots + A_n$ of the n matrices $A_1, A_2, \ldots, A_n$.
  An error message is issued if the given matrices do not have the same dimension.
- The matrices must be of the same domain type, otherwise `FAIL` is returned.

Method `_subtract`: subtract two matrices

```
_subtract(dom A, dom B)
```

- The matrix $A - B$ is returned.

Method `equal`: test on equality of matrices

```
equal(dom A, dom B)
```

- This method tests if the two matrices `A` and `B` are equal and returns `TRUE`, `FALSE` or `UNKNOWN`, respectively.

Method `identity`: identity matrix

```
identity(positive integer n)
```

- This method returns the $n \times n$ identity matrix.
- It only exists if `R` is of category `Cat::Ring`, i.e., a ring with unit.

Method `iszero`: test on zero matrices

```
iszero(dom A)
```

- This method checks whether `A` is a zero matrix and returns `TRUE` or `FALSE`, respectively.
- Note that there may be more than one representation of the zero matrix of a given dimension if `R` does not have the axiom `Ax::canonicalRep`.

Method `transpose`: transpose of a matrix

```
transpose(dom A)
```

- This method returns the tranposed matrix A^t of `A`.

Access Methods

Method `col`: extracting columns

```
col(dom A, column index c)
```

- This method extracts the column with index `c` of the matrix `A` and returns it as a column vector, i.e., a nrows(A) $\times$ 1 matrix.

Method `concatMatrix`: appending of matrices horizontally

```
concatMatrix(dom A, dom B)
```

- This method appends the matrix `B` to the right side of the matrix `A`.
- An error message is issued if the two matrices do not have the same number of rows.

Method `delCol`: deleting columns

```
delCol(dom A, column index c)
```

- This method returns the matrix obtained by deleting the column with index `c` of the matrix `A`.
- If `A` only consists of one column then `NIL` is returned.

Method `delRow`: deleting rows

```
delRow(dom A, row index r)
```

- This method returns the matrix obtained by deleting the row with index `r` of the matrix `A`.
- If `A` only consists of one row then `NIL` is returned.

Method `row`: extracting rows

```
row(dom A, row index r)
```

- This method extracts the row with index `r` of the matrix `A` and returns it as a row vector, i.e., a 1 $\times$ ncols(A) matrix.

Method `setCol`: replacing columns

```
setCol(dom A, column index c, dom v)
```

- This method replaces the column with index `c` of the matrix `A` by the column vector `v`. The vector `v` must be a nrows(A) $\times$ 1 matrix.

Method `setRow`: replacing rows

```
setRow(dom A, row index r, dom v)
```

- This method replaces the row with index `r` of the matrix `A` by the row vector `v`. The vector `v` must be a 1 $\times$ ncols(A) matrix.

Method `stackMatrix`: appending of matrices vertically

```
stackMatrix(dom A, dom B)
```

- This method appends the matrix `B` to the lower end of the matrix `A`.
- An error message is issued if the two matrices do not have the same number of columns.

Method `swapCol`: swapping matrix columns

```
swapCol(dom A, column indices c1, c2)
```

- Returns the matrix which results from swapping the column with index `c1` with the column with index `c2` of `A`.

Method `swapRow`: swapping matrix rows

```
swapRow(dom A, row indices r1, r2)
```

- Returns the matrix which results from swapping the row with index `r1` with the row with index `r2` of `A`.

Changes:

- `Cat::Matrix` used to be `Cat::MatrixCat`.
- The method `"dimen"` was renamed to `"matdim"`.
- The method `"create"` was removed (in `Cat::Matrix` the default implementation was set to the method `"new"`). You may implement this method in the corresponding domain if necessary.

`Cat::Module` – the category of R-modules

`Cat::Module(R)` represents the category of R-modules.

Generating the category:

- `Cat::Module(R)`

Parameters:

R — A domain which must be from the category `Cat::CommutativeRing`.

Categories:

`Cat::LeftModule(R)`, `Cat::RightModule(R)`

Details:

- A `Cat::Module(R)` is a left and right R-module over a commutative ring R.
- Right and left multiplications must be both implemented by the method `"_mult"`.

Changes:

- No changes.

`Cat::Monoid` – the category of monoids

`Cat::Monoid` represents the category of monoids.

Generating the category:

- `Cat::Monoid`

Categories:

`Cat::SemiGroup`

Details:

- A `Cat::Monoid` is a non-abelian semi-group with a neutral element one (`dom::one`) according to the group operation `*` (`_mult`).

Basic Entries:

`one` Must hold the neutral element according to the operation `*`.

Basic Methods

Method `_invert`: returns inverse

```
_invert(dom x)
```

- Must return an inverse of `x` according to the operation `*` or `FAIL` if no inverse exists.

Mathematical Methods

Method `isone`: tests if element is one

```
isone(dom x)
```

- Tests if `x` is equal to one. Uses the method `"equal"` if this domain has not the axiom `Ax::normalRep`.

Method `_power`: tests if element is one

```
_power(dom x, DOM_INT n)
```

- Returns `dom::one` if `n` is 0 and the `n`-fold product of `x` if `n` is positive. If `n` is negative then `x` is inverted. If no inverse exists `FAIL` is returned, otherwise the -`n`-fold product of the inverse.
- This implementation does "repeated squaring".

Changes:

- No changes.

`Cat::OrderedSet` – the category of ordered sets

`Cat::OrderedSet` represents the category of ordered sets.

Generating the category:

- `Cat::OrderedSet`

Categories:

`Cat::BaseCategory`

Details:

- An `Cat::OrderedSet` is a set with a (complete) order relation < (`_less`).
- Use the axiom `Ax::canonicalOrder` to state that elements of a domain are canonically ordered as MuPAD expressions (i.e. ordered with respect to the kernel function `_less`).

Basic Methods

Method `_less`: compares if element is less

`_less(`*dom* `x, `*dom* `y)`

- Must return `TRUE` if `x` is less than `y`.
- An implementation is provided if this domain has axiom `Ax::canonicalOrder`.

Mathematical Methods

Method `_leequal`: compares if element is less or equal

`_leequal(`*dom* `x, `*dom* `y)`

- Returns `TRUE` if `x` is less than or equal to `y`.
- The implementation provided uses the methods `"_less"` and `"equal"`.

Method `max`: returns maximum

`max(`*dom* `x, ...)`

- Returns the maximum of its arguments.

Method `min`: returns minimum

`min(`*dom* `x, ...)`

- Returns the minimum of its arguments.

Method `sort`: sort list of elements

```
sort(Type::ListOf(dom) l)
```

- Sorts the elements of list `l` in ascending order.

Changes:

- No changes.

`Cat::PartialDifferentialRing` – the category of partial differential rings

`Cat::PartialDifferentialRing` represents the category of partial differential rings.

Generating the category:

- `Cat::PartialDifferentialRing`

Categories:

`Cat::CommutativeRing`

Details:

- A `Cat::PartialDifferentialRing` is a commutative ring with a finite set of derivation operators `D_i`.
- A derivation is a linear operator with product rule, i.e. `D_i(f * g)` equals `D_i(f) * g + f * D_i(g)` for all `f` and `g`.
- For many partial differential rings the derivations are differentiations with respect to some indeterminates. Thus in order to support a natural notion it is also supposed that a method `"diff"` exists, such that `diff(f, x)` returns the partial derivation of `f` with respect to the indeterminate `x`.

Basic Methods

Method `D`: returns derivative

```
D(Type::ListOf(Type::PosInt) l, dom x)
```

- Must return the derivative of `x` which is given by the indices in `l`:

  1. If `l` is empty then `x` must be returned.

2. If `l` contains one integer `i` then the `i`-th derivative `D_i(x)` must be returned. If the `i`-th derivativation does not exist `dom::zero` must be returned.
3. If `l` contains more than one integer `i1`,...,`in` than the derivative `D_i1(...D_in(x)...)` must be returned.

Method `diff`: returns partial derivative

`diff(`*`dom`* `x <,` *`variable`* `v, ...>)`

- Must return the derivative of `x` with respect to the variables `v`:

  1. `diff(x)` must return `x`.
  2. `diff(x, v)` must return the partial derivative of `x` with respect to `v`.
  3. `diff(x, v1,..., vn)` must return `diff(...diff(x, v1),...vn)`.

Changes:

- No changes.

`Cat::Polynomial` – the category of multivariate polynomials

`Cat::Polynomial(R)` represents the category of multivariate polynomials over `R`.

Generating the category:

- `Cat::Polynomial(R)`

Parameters:

`R` — A domain which must be from the category `Cat::CommutativeRing`.

Categories:

if `R` is a `Cat::FactorialDomain` then
`Cat::FactorialDomain`

if `R` is a `Cat::GcdDomain` then
`Cat::GcdDomain`

if `R` is a `Cat::IntegralDomain` then
`Cat::IntegralDomain`

`Cat::PartialDifferentialRing`, `Cat::Algebra(R)`

Axioms

if `R` has `Ax::canonicalUnitNormal` then
`Ax::canonicalUnitNormal`

if `R` has `Ax::closedUnitNormals` then
`Ax::closedUnitNormals`

Details:

- A `Cat::Polynomial(R)` is a multivariate polynomial over a commutative coefficient ring `R`.

Entries:

`coeffRing` The coefficient ring `R`.

`characteristic` The characteristic of this domain, which is is the same as that of the ring `R`.

Basic Methods

Method `coeff`: returns coefficients

`coeff(`*`dom`* `p)`

- Must return an expression sequence with the coefficients of `p`.

`coeff(`*`dom`* `p,` *`indeterminate`* `x,` *`Type::NonNegInt`* `n)`

- Must return the coefficient of `x^n` of `p`, which is a polynomial in the remaining indeterminates.

`coeff(`*`dom`* `p,` *`Type::NonNegInt`* `n)`

- Must return the coefficient of `x^n` of `p`, where `x` is the main variable of `p`.

Method `degree`: returns total degree

`degree(`*`dom`* `p)`

- Must return the total degree of `p`.

`degree(`*`dom`* `p,` *`indeterminate`* `x)`

- Must return the degree of `p` with respect to the indeterminate `x`.

Method `degreevec`: returns degree vector

```
degreevec(dom p)
```

- Must return a list with the exponents of the leading term of `p`. The order of the exponents corresponds to the order of the indeterminates as given by the method `"indets"`.

Method `evalp`: evaluates at a point

```
evalp(dom p, indeterminate x = R v, ...)
```

- Must evaluate `p` at the point `x = v` where `x` is an indeterminate and `v` an element of `R`.
- More than one evaluation point may be given. The result must be a polynomial in the remaining indeterminates or an element of `R`.

Method `indets`: returns indeterminates

```
indets(dom p)
```

- Must return a list with the indeterminates of `p`.

Method `lcoeff`: returns leading coefficient

```
lcoeff(dom p)
```

- Must return the leading coefficient of `p`.

Method `lmonomial`: returns leading monomial

```
lmonomial(dom p)
```

- Must return the leading monomial of `p`.

Method `lterm`: returns leading term

```
lterm(dom p)
```

- Must return the leading term of `p`.

Method `mainvar`: returns main variable

```
mainvar(dom p)
```

- Must return the main variable of `p`, which is the first of the indeterminates as given by the method `"indets"`.

Method `mapcoeffs`: map coefficients

```
mapcoeffs(dom p, function f <, a, ...>)
```

- Must replace the coefficients `c_i` of `p` by the results of the function calls `f(c_i, a, ...)`.

Method `multcoeffs`: multiply coefficients

```
multcoeffs(dom p, R c)
```

- Must multiply all coefficients of `p` by `c`.

Method `nterms`: return number of terms

```
nterms(dom p)
```

- Must return the number of non-zero terms of `p`.

Method `nthcoeff`: return n-th coefficient

```
nthcoeff(dom p, Type::PosInt n)
```

- Must return the `n`-th coefficient of `p`.

Method `nthmonomial`: return n-th monomial

```
nthmonomial(dom p, Type::PosInt n)
```

- Must return the `n`-th monomial of `p`.

Method `nthterm`: return n-th term

```
nthterm(dom p, Type::PosInt n)
```

- Must return the `n`-th term of `p`.

Method `tcoeff`: return trailing coefficient

`tcoeff(`*dom* `p)`

- Must return the trailing (i.e. last) coefficient of `p`.

Method `unitNormal`: returns unit normal

`unitNormal(`*dom* `p)`

- Must return the unit normal representation of `p`.
- An implementation is provided if `R` has the axiom `Ax::canonicalUnitNormal`: In this case `p` is multiplied by an unit of `R` such that the leading coefficient has unit normal representation in `R`.

Method `unitNormalRep`: returns unit normal representation

`unitNormalRep(`*dom* `p)`

- Must return the unit normal representation of `p` and the factors needed to bring `p` into unit normal form (see `Cat::IntegralDomain` for the return value expected).
- An implementation is provided if `R` has the axiom `Ax::canonicalUnitNormal`.

Mathematical Methods

Method `content`: return content

`content(`*dom* `p)`

- Returns the content of `p` if `R` is a `Cat::GcdDomain`.

Method `isUnit`: tests if element is a unit

`isUnit(`*dom* `p)`

- Returns `TRUE` iff `p` is a unit.

Method `primpart`: returns primitive part

`primpart(`*dom* `p)`

- Returns the primitive part of `p` if `R` is a `Cat::GcdDomain`: The content of `p` is removed and the unit normal of the result is returned.

Method `solve`: solves polynomial equation

`solve(`*`dom`* `p, `*`indeterminate`* `x <, opt, ...>)`

- Solves the polynomial equation `p = 0` with respect to `x` over the domain `R`. See the function `solve` for details about the optional arguments `opt, ...`.

`solve(`*`dom`* `p, `*`indeterminate`* `x = `*`DOM_DOMAIN`* `T <, opt, ...>)`

- Solves the polynomial equation `p = 0` with respect to `x` over the domain `T`. See the function `solve` for details about the optional arguments `opt, ...`.

`solve(`*`dom`* `p)`

- The polynomial `p` must be univariate. Solves the polynomial equation `p = 0` with respect to the indeterminate of `p` over the domain `R`.

Changes:

- Has been renamed. Used to be `Cat::PolynomialCat`.

`Cat::PrincipalIdealDomain` – the category of principal ideal domains

`Cat::PrincipalIdealDomain` represents the category of principal ideal domains.

Generating the category:

- `Cat::PrincipalIdealDomain`

Categories:

`Cat::GcdDomain`

Details:

- A `Cat::PrincipalIdealDomain` is an integral domain with gcd where each ideal is principal. Note that the method `"idealGenerator"` has to find generators for finitely generated ideals only.

Basic Methods

Method `idealGenerator`: return generator of ideal

```
idealGenerator(dom x, ...)
```

- Must return the generator of the ideal generated by its arguments.

Changes:

- No changes.

`Cat::QuotientField` – the category of quotient fields

`Cat::QuotientField(R)` represents the category of quotient fields over `R`.

Generating the category:

- `Cat::QuotientField(R)`

Parameters:

`R` — A domain which must be from the category `Cat::IntegralDomain`.

Categories:

`Cat::Field`, `Cat::Algebra(R)`,

if `R` has `Cat::OrderedSet` then
`Cat::OrderedSet`

Details:

- A `Cat::QuotientField` is the field of fractions over the integral domain `R`.

Entries:

`characteristic` The characteristic of this domain, which is is the same as that of `R`.

Basic Methods

Method `denom`: return denominator

`denom(`*`dom`* `x)`

- Must return the denominator of `x`, which is an element of `R`.

Method `numer`: return numerator

`numer(`*`dom`* `x)`

- Must return the numerator of `x`, which is an element of `R`.

Mathematical Methods

Method `equal`: test for equality

`equal(`*`dom`* `x, `*`dom`* `y)`

- Implements an equality test by testing if the cross-product of numerators and denominators are equal in `R`.

Method `iszero`: test for zero

`iszero(`*`dom`* `x)`

- Implements a test for zero by testing if the numerator is zero in `R`.

Method `_less`: test if element is less

`_less(`*`dom`* `x, `*`dom`* `y)`

- Implements an ordering if the domain `R` has the axiom `Cat::OrderedSet`: If `R` is ordered then this method implements an ordering which is given by the ordering of the cross-product of numerators and denominators in `R`.

Method `retract`: returns retracted element

`retract(`*`dom`* `x)`

- If `x` may be "retracted" to an element of `R` (i.e. if the factor `x` may be regarded as an element of `R`) this element is returned, otherwise `FAIL` is returned.
- The default implementation uses the method `"_divide"` to divide numerator and denominator.

Changes:

- No changes.

`Cat::RightModule` – the category of right R-modules

`Cat::RightModule(R)` represents the caregory of right `R`-modules.

Generating the category:

- `Cat::RightModule(R)`

Parameters:

`R` — A domain which must be from the category `Cat::Ring`.

Categories:

`Cat::AbelianGroup`

Details:

- A `Cat::RightModule` is an abelian group together with a ring `R` and a right multiplication `*` (`_mult`).
- The right multiplication is an operation taking an element of ring `R` and a module element and returning a module element.
- Given ring elements a, b and module elements x, y the following 3 distibutive laws must hold:

  1. $x(ab) = (xa)b$,
  2. $x(a + b) = xa + xb$,
  3. $(x + y)a = xa + ya$.

- Beware: The operation of a non-abelian semi-group is also written as `*` (`_mult`). The method `"_mult"` must handle the situation if a right module is also a non-abelian semi-group. In such a case it must both implement the group operation and the right multiplication by elements of the ring.

Basic Methods

Method `_mult`: right multiplication by a ring element

```
_mult(dom x, R r)
```

- Must return the right multiplication of `x` by the ring element `r`.

Changes:

- No changes.

`Cat::Ring` – the category of rings

`Cat::Ring` represents the category of rings.

Generating the category:

- `Cat::Ring`

Categories:

`Cat::Rng`, `Cat::Monoid`, `Cat::LeftModule(dom)`

Details:

- A `Cat::Ring` is a ring with a unit `dom::one`, i.e. an abelian group according to the operation `+` (`_plus`) and a non-abelian monoid according to the operation `*` (`_mult`) where in addition the two distributive laws $a(b+c) = ab + ac$ and $(a+b)c = ac + bc$ hold.
- A `Cat::Ring` is also a left module over itself. The left multiplication of the module is also writen as `*` (`_mult`).
- Note that a ring without unit is a `Cat::Rng`.

Basic Entries:

`characteristic` Must hold the characteristic of this ring.

Changes:

- No changes.

`Cat::Rng` – the category of rings without unit

`Cat::Rng` represents the category of rings without unit.

Generating the category:

- `Cat::Rng`

Categories:

`Cat::AbelianGroup`, `Cat::SemiGroup`

Details:

- A `Cat::Rng` is a ring without a unit, i.e. an abelian group according to the operation `+` (`_plus`) and a non-abelian semi-group according to the operation `*` (`_mult`) where in addition the two distributive laws $a(b+c) = ab + ac$ and $(a+b)c = ac + bc$ hold.
- Use the axiom `Ax::noZeroDivisors` to state that there are no zero divisors according to `*`, i.e. that the product of non-zero elements never is zero.

Changes:

- No changes.

`Cat::SemiGroup` – the category of semi-groups

`Cat::SemiGroup` represents the category of semi-groups.

Generating the category:

- `Cat::SemiGroup`

Categories:

`Cat::BaseCategory`

Details:

- A `Cat::SemiGroup` represents the category of non-abelian semi-groups, where the group operation is written as multiplication. Hence a `Cat::SemiGroup` is a set with an associative operation `*` (`_mult`).
- Note that abelian semi-groups with operation `+` have category `Cat::AbelianSemiGroup`.

Basic Methods

Method `_mult`: returns product

```
_mult(dom x, ...)
```

- Must return the product of its arguments.

Mathematical Methods

Method `_power`: returns power

```
_power(dom x, Type::PosInt n)
```

- Returns the `n`-fold product of `x`. The implementation provided does "repeated squaring".

Changes:

- No changes.

`Cat::Set` – the category of sets of complex numbers

`Cat::Set` represents the category of subsets of the complex numbers.

Sets of this category allow set–theoretic operations as well as pointwise arithmetical operations.

Generating the category:

- `Cat::Set`

Categories:

`Cat::BaseCategory`

Details:

- The main feature of `Cat::Set` is a particular overloading mechanism. It provides *n*-ary operators that can handle operands from different domains of category `Cat::Set`, as well as mixed input where some operands are of types not belonging to `Cat::Set`. *Hence, in the methods of* `Cat::Set`*, operands of arbitrary type are allowed.*
- There are three kinds of operators: *n*-ary (associative and commutative), binary (not assumed to be commutative), and unary (mapping a function). `Cat::Set` provides generic methods for generating these kinds of operators, and uses them to define default methods overloading the common set–theoretic and arithmetical functions.
- By default, any operation of sets is defined, but returns unevaluated since the arithmetical or set–theoretic expression cannot be simplified. Each domain of type `Cat::Set` must provide particular slots and tables in order to achieve simplifications in certain special cases.
- Arithmetical operations are defined pointwise. It is not an error if some operation is not defined for all elements of a set.
- `Cat::Set` is mainly used by domains of sets returned by `solve`.

Mathematical Methods

Method `commassop`: returns an n-ary commutative and associative operator for sets

```
commassop(string operatorname)
```

- This method returns a procedure that applies the law of composition specified by `operatorname`, by searching applicable methods in the domains the operands belong to.
- The returned procedure first sorts its operands (which it may do because of commutativity). Those operands not belonging to a domain of category `Cat::Set` are handled by the usual overloading mechanism, i.e. by the slot `operatorname` of one of their domains. Out of the others, several operands belonging to the same domain are handled by the slot `"homog".operatorname` of that domain. Finally, the returned method tries to combine each possible pair of operands. If they are from the same domain, `"bin".operatorname` is called for them. The following is done if the operands are from different domains: let `T1` and T2 be their types; then their `"inhomog".operatorname` slots are used. If such a slot exists in the domain `T1`, it must contain a table indexed by possible types `T2`, and the entry at that index must be a procedure that carries out the

operation for exactly two arguments, the first being a `T1`, the second being a `T2`. Conversely, if such a slot exists in the domain `T2`, it must contain a table indexed by possible types `T1`, and the entry at that index must be a procedure that carries out the operation for exactly two arguments, the first being a `T2`, the second being a `T1`.

- The slot `"homog ".operatorname`, or a table entry in the slot `"inhomog".operatorname`, may return `FAIL` in order to indicate that it could not simplify its input; if they are missing, this indicates that a simplification is generally not possible for input of this type. In these cases, the returned procedure proceeds by trying to combine another two of the given arguments.
- A slot `"bin".operatorname` usually won't exist, except for the case that there is no `"homog".operatorname`; usually the latter can also take care for the case of exactly two operands.
- The whole process is repeated over and over until no new simplifications occur or only one operand is left. If no more simplifications occur, an unevaluated call to the operator is returned, the arguments being all remaining operands that could not be combined further.

Method `binop`: returns a binary operator for sets

`binop(`*`string`* `operatorname)`

- This method returns a procedure that applies the law of composition specified by `operatorname`, by searching applicable methods in the domains the operands belong to.
- The returned procedure uses the slot `"bin".operatorname` of its first argument if both arguments are of the same type. Otherwise it uses the slot `"inhomogleft".operatorname` of its first argument; if that fails, it uses the slot `"inhomogright".operatorname` of its second argument; each of these slots, if it exists, must contain tables, indexed by the type of the other argument, such that `slot(T1, "inhomogleft".operatorname)[T2]` and `slot(T2, "inhomogright".operatorname)[T1]` carry out the operation for objects of type `T1` and `T2`, in this order.
- No commutativity of the operation is assumed.
- If the slots or table entries do not exist or return `FAIL`, an unevaluated call to the operator is returned.

Method `homogassop`: returns a n-ary operator for sets belonging to the same domain

`homogassop(`*`string`* `operatorname)`

- This method returns a procedure which simply splits its argument sequence into two parts and calls itself recursively for both halves, thereby reducing the n-ary operation to several binary operations. These binary operations are carried out by the slot `"bin".operatorname` of the domain all operands stem from.

Method `_union`: union of sets

```
_union(any S1, ...)
```

- The union of sets is computed by the commutative-associative operator generated by `"commassop"`, using the slots `"homog_union"` and `"inhomog_union"` of the domains of its operands.

Method `_intersect`: intersection of sets

```
_intersect(any S1, ...)
```

- The intersection of sets is computed by the commutative-associative operator generated by `"commassop"`, using the slots `"homog_intersect"` and `"inhomog_intersect"` of the domains of its operands.

Method `_plus`: set of sums of set elements

```
_plus(any S1, ...)
```

- The sum $S_1 + \ldots + S_k$ is defined to be the set of all sums $s_1 + \ldots + s_k$, with $s_i \in S_i$.
- The sum of sets is computed by the commutative-associative operator generated by `"commassop"`, using the slots `"homog_plus"` and `"inhomog_plus"` of the domains of its operands.

Method `_mult`: set of product of set elements

```
_mult(any S1, ...)
```

- The product $S_1 * \cdots * S_k$ is defined to be the set of all products $s_1 * \cdots * s_k$, with $s_i \in S_i$.
- The product of sets is computed by the commutative-associative operator generated by `"commassop"`, using the slots `"homog_mult"` and `"inhomog_mult` of the domains of its operands.

Method `_minus`: difference set

```
_minus(any S1, any S2)
```

- The difference of sets is computed by the binary operator generated by `"binop"`, using the slots `"homog_minus"`, `"inhomogleft_minus"`, and `"inhomogright_minus"` of its operands.

Method `_power`: pointwise power

```
_power(any S1, any S2)
```

- The power $S_1^{S_2}$ is defined to be the set of all powers $s_1^{s_2}$, where $s_i \in S_i$.
- The power of sets is computed by the binary operator generated by `"binop"`, using the slots `"homog_power"`, `"inhomogleft_power"`, and `"inhomogright_power"` of its operands.

Method `map`: map an operation to a set

```
map(any S, any f)
```

- This method returns the set $\{f(x); x \in S\}$, which is of type `Dom::ImageSet`.
- By overloading this method in a particular domain, the behaviour of sets changes whenever a special function is applied to them.

Changes:

- `Cat::Set` is a new function.

`Cat::SkewField` – the category of skew fields

`Cat::SkewField` represents the category of skew fields (division rings).

Generating the category:

- `Cat::SkewField`

Categories:

`Cat::Ring`

Details:

- A `Cat::SkewField` represents a ring with unit where each nonzero element is invertible. This structure is also called division ring in the literature.

Changes:

- No changes.

`Cat::SquareMatrix` – the category of square matrices

`Cat::SquareMatrix(R)` represents the category of square matrices over the rng `R`.

Generating the category:

- `Cat::SquareMatrix(R)`

Parameters:

`R` — A domain which must be from the category `Cat::Rng`.

Categories:

if `R` has `Cat::Ring` then
`Cat::Ring`

`Cat::Rng`, `Cat::Matrix(R)`

Details:

- A `Cat::SquareMatrix(R)` represents the rng (ring without unit) of square matrices over the coefficient domain `R`.

Entries:

`characteristic` Defined if `R` is a ring: In this case the characteristic of the matrix domain is the same as that of `R`.

Changes:

- `Cat::SquareMatrix` used to be `Cat::SquareMatrixCat`.

`Cat::UnivariatePolynomial` – the category of univariate polynomials

`Cat::UnivariatePolynomial(R)` represents the category of univariate polynomials over `R`.

Generating the category:

- `Cat::UnivariatePolynomial(R)`

Parameters:

`R` — A domain which must be from the category `Cat::CommutativeRing`.

Categories:

if `R` has `Cat::Field` then
`Cat::EuclideanDomain`

`Cat::Polynomial(R)`, `Cat::DifferentialRing`

Details:

- A `Cat::UnivariatePolynomial(R)` is a univariate polynomial over the commutative ring `R`.

Basic Methods

Method `pdivide`: pseudo-divide polynomials

`pdivide(`*`dom`* `p, `*`dom`* `q)`

- Must compute the pseudo-division of `p` and `q`.
- Must return a sequence `(b, s, r)` of a ring element `b` and polynomials `s` and `r` such that `multcoeffs(p, b) = s * q + r` holds with `b = lcoeff(q)^(degree(p) - degree(q) + 1)`.

Method `pquo`: returns pseudo-quotient

```
pquo(dom p, dom q)
```

- Must return the pseudo-quotient of `p` and `q`, i.e. the second element `s` of the sequence returned by the method `"pdivide"`.

Method `prem`: returns pseudo-reminder

```
prem(dom p, dom q)
```

- Must return the pseudo-reminder of `p` and `q`, i.e. the third element `r` of the sequence returned by the method `"pdivide"`.

Changes:

- Has been renamed. Used to be `Cat::UnivariatePolynomialCat`.

`Cat::VectorSpace` – the category of vector spaces

`Cat::VectorSpace(F)` represents the category of vector spaces over the field `F`.

Generating the category:

- `Cat::VectorSpace(F)`

Parameters:

`F` — A domain which must be from the category `Cat::Field`.

Categories:

`Cat::Module(F)`

Details:

- A `Cat::VectorSpace(F)` represents the category of vector spaces over the field `F`. A vector space is a abelian group with an operation `+` (`_plus`).
- The scalar product has to be implemented via the method `"_mult"`. Other kinds of multiplication are not defined.

Basic Methods

Method `_mult`: returns scalar product

`_mult(`*`F`* `c, `*`dom`* `x)`

- Must return the scalar product of `c` and `x`.

`_mult(`*`dom`* `x, `*`F`* `c)`

- Must return the scalar product of `x` and `c`.

Changes:

- No changes.