Type — library for type checking and mathematical properties

Table of contents

Preface iii

`Type::AlgebraicConstant` — a type representing algebraic constants 1

`Type::AnyType` — a type representing arbitrary MuPAD objects . 2

`Type::Arithmetical` — a type representing arithmetical objects 3

`Type::Complex` — a type and a property representing complex numbers 4

`Type::Constant` — a type representing constant objects 6

`Type::ConstantIdents` — set of constant identifiers in MuPAD . 8

`Type::Equation` — a type representing equations 10

`Type::Even` — a type and a property representing even integers . 11

`Type::Imaginary` — a type and a property representing imaginary numbers 13

`Type::IndepOf` — a type representing objects that do not contain given identifiers 14

`Type::Integer` — a type and a property representing integers . . 16

`Type::Interval` — a property representing intervals 17

`Type::ListOf` — type for testing lists of objects with the same type 20

`Type::ListProduct` — type for testing lists 21

`Type::NegInt` — a type and a property representing negative integers 23

`Type::NegRat` — a type and a property representing negative rational numbers 25

`Type::Negative` — a type and a property representing negative numbers 26

`Type::NonNegInt` — a type and a property representing nonnegative integers 28

`Type::NonNegRat` — a type and a property representing nonnegative rational numbers 30

`Type::NonNegative` — a type and a property representing nonnegative numbers 32

`Type::NonZero` — a type and a property representing "unequal to zero" 34

`Type::Numeric` — a type for testing numerical objects 36

`Type::Odd` — a type and a property representing odd integers . . 37

`Type::PolyExpr` — type for testing polynomial expressions . . . 39

`Type::PolyOf` — type for testing polynomials 41

`Type::PosInt` — a type and a property representing positive integers 42

`Type::PosRat` — a type and a property representing positive rational numbers 44

`Type::Positive` — a type and a property representing positive numbers 45

`Type::Prime` — a type and a property representing prime numbers 47

`Type::Product` — type for testing sequences 49

`Type::Property` — type to identify properties 50

`Type::RatExpr` — type for testing rational expressions 52

`Type::Rational` — a type and a property representing rational numbers 53

`Type::Real` — a type and a property representing real numbers . 55

`Type::Relation` — type for testing relations 57

`Type::Residue` — a property representing a residue class 58

`Type::SequenceOf` — type for testing sequences 60

`Type::Series` — type for testing series 62

`Type::SetOf` — type for testing sets 64

`Type::Singleton` — type to identify exactly one object 65

`Type::TableOfEntry` — type for testing tables with specified entries 67

`Type::TableOfIndex` — type for testing tables with specified indices 68

`Type::Union` — type for testing several types with one call 69

`Type::Unknown` — type for testing variables 71

`Type::Zero` — a type and a property representing zero 72

The library Type

This library contains several objects to perform syntactical tests with `testtype` (see example 1).

Some of the objects depends of arguments, that must be given by the user (see example 2).

Some of the objects can be used as mathematical properties within `assume` and `is` (see example 3).

All other objects that are not properties cannot be used within `assume` and `is` (see example 4).

The next tables gives an overview of all objects in this library:

Name	syntactical test	is a property	has arguments
Type::AlgebraicConstant	yes	no	no
Type::AnyType	yes	no	no
Type::Arithmetical	yes	no	no
Type::Complex	yes	yes	no
Type::Constant	yes	no	no
Type::ConstantIdents	yes	no	no
Type::Equation	yes	no	yes
Type::Even	yes	yes	no
Type::Function	yes	no	no
Type::Imaginary	yes	yes	no
Type::IndepOf	yes	no	yes
Type::Integer	yes	yes	no
Type::Interval	no	yes	yes
Type::ListOf	yes	no	yes
Type::ListProduct	yes	no	yes
Type::NegInt	yes	yes	no
Type::NegRat	yes	yes	no
Type::Negative	yes	yes	no
Type::NonNegInt	yes	yes	no
Type::NonNegRat	yes	yes	no
Type::NonNegative	yes	yes	no
Type::NonZero	yes	yes	no
Type::Numeric	yes	no	no
Type::Odd	yes	yes	no
Type::PolyOf	yes	no	yes
Type::PosInt	yes	yes	no
Type::PosRat	yes	yes	no
Type::Positive	yes	yes	no
Type::Prime	yes	yes	no
Type::Product	yes	no	yes
Type::Property	yes	no	no
Type::Rational	yes	yes	no
Type::Real	yes	yes	no

Type::Relation	yes	no	no
Type::Residue	yes	yes	yes
Type::SequenceOf	yes	no	yes
Type::Series	yes	no	yes
Type::SetOf	yes	no	yes
Type::Singleton	yes	no	no
Type::TableOfEntry	yes	no	yes
Type::TableOfIndex	yes	no	yes
Type::Union	yes	no	yes
Type::Unknown	yes	no	no
Type::Zero	yes	yes	no

Example 1. `testtype` performs syntactical tests:

```
>> testtype([1, 2, 3], Type::ListOf(Type::PosInt))

                                TRUE

>> testtype(3 + 4*I, Type::Constant)

                                TRUE
```

Example 2. Some types depends on parameters and cannot be used without parameters:

```
>> testtype([1, 2, 3], Type::ListOf)

                                FALSE

>> testtype(x = 0, Type::Equation(Type::Unknown, Type::Zero))

                                TRUE
```

An interval must be given with borders, otherwise it is not a property:

```
>> assume(x, Type::Interval)

 Error: second argument must be a property [property::assume]

>> assume(x, Type::Interval(0, infinity))

                    ]0, infinity[ of Type::Real
```

Example 3. `is` derives mathematical properties:

```
>> assume(x > 0):
   is(sqrt(x^2), Type::NonNegative)

                                TRUE

>> is(-(2*x + 1) < 0)

                                TRUE
```

Example 4. `Type::Property` and `Type::Constant` are not properties:

```
>> assume(x, Type::Property)

 Error: second argument must be a property [property::assume]

>> is(x, Type::AnyType)

 Error: wrong type of second argument 'Type::AnyType' ('Type::P\
 roperty' expected) [property::is]
```

`Type::AlgebraicConstant` – a type representing algebraic constants

`Type::AlgebraicConstant` represents algebraic constants.

Call(s):

- `testtype(obj, Type::AlgebraicConstant)`

Parameters:

`obj` — any MuPAD object

Return Value: see `testtype`

Related Functions: `testtype`, `Type::Constant`

Details:

- In MuPAD, algebraic constants are characterized as follows: a complex number is an algebraic constant, if both its real part and its imaginary part are rational. Sums and products of algebraic constants are again algebraic constants. Further, rational powers of algebraic constants are again algebraic constants.

  Taken together, these rules characterize algebraic constants over the rationals defined as usual, i.e., as roots of polynomial expressions.

- This type does not represent a property: it cannot be used in `assume` to mark an identifier as an algebraic constant.

Example 1. The following number is composed of radicals involving rational numbers and therefore is an algebraic constant:

```
>> testtype((3^(1/2)*I + 1/8)^(1/7), Type::AlgebraicConstant)
```

```
TRUE
```

The following objects are not algebraic constants:

```
>> testtype(2^I, Type::AlgebraicConstant),
   testtype(PI, Type::AlgebraicConstant)
```

```
FALSE, FALSE
```

Example 2. Symbolic objects cannot represent algebraic constants:

```
>> testtype(x, Type::AlgebraicConstant)
```

```
FALSE
```

Example 3. The following call selects the algebraic constants in an expression:

```
>> select(x + PI + 2^(1/2) + I, testtype, Type::AlgebraicConstant)
```

```
  1/2
2    + I
```

Changes:

- No changes.

`Type::AnyType` – a type representing arbitrary MuPAD objects

`Type::AnyType` represents arbitrary MuPAD objects.

Call(s):

- `testtype(obj, Type::AnyType)`

Parameters:

`obj` — any MuPAD object

Return Value: `testtype` always returns `TRUE`

Related Functions: `testtype`

Details:

- This type is meant to represent arbitrary MuPAD objects in constructors of composite types such as `Type::ListOf`.
- This type does not represent a property: it cannot be used in `assume`.

Example 1. Any object matches this type:

```
>> testtype(3, Type::AnyType),
   testtype(x, Type::AnyType),
   testtype(array(1..1, [x]), Type::AnyType),
   testtype(Dom::Matrix(), Type::AnyType)

                    TRUE, TRUE, TRUE, TRUE
```

This type is meant for constructing composite types. The following call tests, whether an object is a list with arbitrary elements:

```
>> testtype([3, x, array(1..1, [x]), Dom::Matrix()],
            Type::ListOf(Type::AnyType))

                                TRUE
```

Changes:

- No changes.

`Type::Arithmetical` – a type representing arithmetical objects

`Type::Arithmetical` represents arithmetical objects.

Call(s):

- `testtype(obj, Type::Arithmetical)`

Parameters:

`obj` — any MuPAD object

Return Value: see `testtype`

Related Functions: `testtype`

Details:

- In MuPAD, arithmetical objects are objects for which arithmetical operations such as addition, multiplication, exponentiation etc. are defined. These include numbers, most expressions, `infinity` and elements of certain library domains. In particular, the latter include `rectform` objects and series expansions of domain type `Series::Puiseux`.

- The following objects are *not* regarded as arithmetical objects:
  - equations and inequalities,
  - Boolean objects and Boolean expressions involving `and`, `or`, `not`,
  - lists,
  - sets and set expressions involving `union`, `intersect`, `minus`,
  - polynomials of domain type `DOM_POLY`,
  - functions and procedures,
  - arrays and tables.
- This type does not represent a property: it cannot be used in `assume` to mark an identifier as an arithmetical object.

Example 1. Numbers and expressions are regarded as arithmetical objects:

```
>> testtype(3 + I, Type::Arithmetical),
   testtype(x + sqrt(2) + I*PI, Type::Arithmetical),
   testtype(x/y + y/x, Type::Arithmetical)

                          TRUE, TRUE, TRUE
```

Equations and inequalities are not regarded as arithmetical objects:

```
>> testtype(x^2 = 2, Type::Arithmetical),
   testtype(x <> 2, Type::Arithmetical),
   testtype(x < 2, Type::Arithmetical),
   testtype(x >= 2, Type::Arithmetical)

                    FALSE, FALSE, FALSE, FALSE
```

Sets, lists, tables and arrays are not arithmetical:

```
>> testtype({a, b, c}, Type::Arithmetical),
   testtype(array(1..1, [x]), Type::Arithmetical)

                            FALSE, FALSE
```

However, domain objects such as matrices of some matrix domain are arithmetical:

```
>> testtype(Dom::Matrix()([[1, 2], [3, 4]]), Type::Arithmetical)

                                TRUE
```

Changes:

- `Type::Arithmetical` is a new function.

`Type::Complex` – a type and a property representing complex numbers

`Type::Complex` represents complex numbers. This type can also be used as a property to mark identifiers as complex numbers.

Call(s):

- `testtype(obj, Type::Complex)`
- `assume(x, Type::Complex)`
- `is(ex, Type::Complex)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier
`ex` — an arithmetical expression

Return Value: see `assume`, `is` and `testtype`

Related Functions: `assume`, `is`, `testtype`, `Type::Imaginary`, `Type::Property`, `Type::Real`

Details:

- The call `testtype(obj, Type::Complex)` checks, whether `obj` is a complex number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` and `DOM_COMPLEX`. This does not include arithmetical expressions such as `exp(1)`, which are not identified as of type `Type::Complex`.
- The call `assume(x, Type::Complex)` marks the identifier `x` as a complex number.

  The call `is(ex, Type::Complex)` derives, whether the expression `ex` is a complex number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::Complex`:

```
>> testtype(2, Type::Complex),
   testtype(3/4, Type::Complex),
   testtype(0.123, Type::Complex),
   testtype(1 + I/3, Type::Complex),
   testtype(1.0 + 2.0*I, Type::Complex)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of complex numbers. Syntactically, however, they are not of type `Type::Complex`:

```
>> testtype(exp(3), Type::Complex),
   testtype(PI^2 + 5, Type::Complex),
   testtype(sin(2) + PI*I, Type::Complex)

                    FALSE, FALSE, FALSE
```

Example 2. Identifiers may be assumed to represent a complex number:

```
>> assume(x, Type::Complex): is(x, Type::Complex)

                              TRUE
```

The real numbers are a subset of the complex numbers:

```
>> assume(x, Type::Real): is(x, Type::Complex)

                              TRUE
```

Without further information, it cannot be decided whether a complex number is real:

```
>> assume(x, Type::Complex): is(x, Type::Real)

                             UNKNOWN

>> unassume(x):
```

Changes:

- No changes.

`Type::Constant` – a type representing constant objects

`Type::Constant` represents constant objects, i.e., objects not containing symbolic identifiers.

Call(s):

- `testtype(obj, Type::Constant)`

Parameters:

`obj` — any MuPAD object

Return Value: see `testtype`

Related Functions: `testtype`

Details:

- Numbers, strings, Boolean constants, points, polygons, `NIL`, `FAIL` and the identifiers `PI`, `EULER` and `CATALAN` in the set `Type::ConstantIdents` are regarded as constant objects. A composite object is constant, if all its operands are constant.
- Any function is identified as a constant, if all arguments are constant, also if the function is not defined (e.g., an identifier).
- This type does not represent a property: it cannot be used in `assume` to mark an identifier as a constant.

Example 1. The following objects are elementary constants:

```
>> testtype(3, Type::Constant),
   testtype(sin(3/2), Type::Constant),
   testtype(TRUE, Type::Constant),
   testtype("MuPAD", Type::Constant),
   testtype(FAIL, Type::Constant)
```

```
                TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expression contains an indeterminate x and, consequently, is not a constant object:

```
>> testtype(exp(x + 1), Type::Constant)
```

```
                                FALSE
```

All constant operands of an expression are selected:

```
>> select(x^2 + 3*x - 2, testtype, Type::Constant)
```

```
                                 -2
```

Any function call is considered constant, if the arguments are constant:

```
>> testtype(f(1, 2, 3, 4), Type::Constant)
```

```
                                TRUE
```

Changes:

- No changes.

`Type::ConstantIdents` – set of constant identifiers in MuPAD

`Type::ConstantIdents` is the set `{PI, EULER, CATALAN}`.

Call(s):

- `contains(Type::ConstantIdents, obj)`

Parameters:

`obj` — any MuPAD object

Return Value: see `contains`

Related Functions: `contains`, `indets`, `Type::Constant`

Details:

- `Type::ConstantIdents` is the set of identifiers that represent constants. As of version 2.0, these are `PI`, `EULER` and `CATALAN`. The constant `E` is not in this set, because MuPAD replaces it directly after the input by `exp(1)`.
- These constants will be returned by the function `indets`, but they cannot be treated like other identifiers. For example, they cannot have properties or be the left-hand side of an assignment.

  See example 1 for an application.
- `Type::Constant` makes use of `Type::ConstantIdents`, see example 2.

Example 1. MuPAD implements π as the identifier `PI`.

```
>> domtype(PI)
                                DOM_IDENT
```

However, `PI` is constant (although rumors keep raising their heads that China, Alabama, or whoever it may be next time had tried to change its value by means of a legislative process):

```
>> testtype(PI, Type::Constant)
```

```
                                TRUE
```

Still, `indets` regards `PI` as an identifier with no value (which is syntactically correct), and you can even use `PI` as an indeterminate of a polynomial:

```
>> indets(PI/2*x);
   poly(PI/2*x)
                              {x, PI}

                       poly(1/2 PI x, [PI, x])
```

To find the "real" indeterminates, use the following call:

```
>> indets(PI/2*x) minus Type::ConstantIdents
                                {x}
```

Example 2. Assume you want MuPAD to regard the identifier `KHINTCHINE` as a constant. (Probably, it should represent the Khintchine constant K, which is approximately 2.685452, but we will not implement this.) First of all, you should make sure that the identifier does not have a value yet and protect it:

```
>> testtype(KHINTCHINE, DOM_IDENT);
   protect(KHINTCHINE, Error)
                                TRUE

                                None
```

Next, add `KHINTCHINE` to `Type::ConstantIdents` (note that we have to unprotect the identifier `Type`, because `Type::ConstantIdents` is a slot of it):

```
>> old_protection := unprotect(Type):
   Type::ConstantIdents := Type::ConstantIdents union {KHINTCHINE}:
   protect(Type, old_protection):
   Type::ConstantIdents
                 {PI, EULER, CATALAN, KHINTCHINE}
```

Now, MuPAD regards `KHINTCHINE` as a constant:

```
>> testtype(sin(PI + KHINTCHINE), Type::Constant)
                                TRUE

>> solve(x^2 = KHINTCHINE)
                           1/2                     1/2
          {[x = KHINTCHINE   ], [x = - KHINTCHINE   ]}
```

Changes:

- The constant `CATALAN` was added.

`Type::Equation` – a type representing equations

`Type::Equation` represents equations. The types of the left hand side and the right hand side can be specified.

Call(s):

- `testtype(obj, Type::Equation(<lhs_type <, rhs_type>>))`

Parameters:

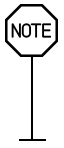
`obj`	—	any MuPAD object
`lhs_type`	—	the type of the left hand side; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`
`rhs_type`	—	the type of the right hand side

Return Value: see `testtype`

Related Functions: `testtype`

Details:

- The call `testtype(obj, Type::Equation(lhs_type,rhs_type))` checks whether `type(obj)` yields `"_equal"` and `testtype(lhs(obj), lhs_type)` and `testtype(rhs(obj), rhs_type)` both yield `TRUE` and returns `TRUE`, if all holds, otherwise `FALSE`.
- The two optional parameters `lhs_type` and `rhs_type` determine the types of the left hand side and the right hand side, respectively.
- The default values of `lhs_type` and `rhs_type` are `Type::AnyType`.
- The equations `lhs=rhs` and `rhs=lhs` are considered different! E.g., the equation `x=3` matches the type `Type::Equation(DOM_IDENT,DOM_INT)`, but it does not match the type `Type::Equation(DOM_INT,DOM_IDENT)`.



- This type does not represent a property, it cannot be used in an `assume` call.

Example 1. The following object is an equation:

```
>> testtype(x = 3, Type::Equation())
```

```
TRUE
```

The following calls test, whether the object is an equation with an unknown on the left hand side and a positive integer on the right hand side:

```
>> testtype(x = 3, Type::Equation(Type::Unknown, Type::PosInt)),
   testtype(x = 0, Type::Equation(Type::Unknown, Type::PosInt))
```

```
TRUE, FALSE
```

Changes:

- `Type::Equation` is a new function.

`Type::Even` – a type and a property representing even integers

`Type::Even` represents even integers. This type can also be used as a property to mark identifiers as even integers.

Call(s):

- `testtype(obj, Type::Even)`
- `assume(x, Type::Even)`
- `is(ex, Type::Even)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `assume`, `is` and `testtype`

Related Functions: `is`, `assume`, `testtype`, `Type::Odd`, `Type::Property`

Details:

- The call `testtype(obj, Type::Even)` checks, whether `obj` is an even number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(domtype(x/2) = DOM_INT)` holds.
- The call `assume(x, Type::Even)` marks the identifier `x` as an even number.

  The call `is(ex, Type::Even)` derives, whether the expression `ex` is an even number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::Even`:

```
>> testtype(2, Type::Even),
   testtype(-4, Type::Even),
   testtype(8, Type::Even),
   testtype(-11114, Type::Even),
   testtype(4185296581467695598, Type::Even)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. We use this type as a property:

```
>> assume(x, Type::Even):
```

The following calls to `is` derive the properties of a composite expression from the properties of its indeterminates:

```
>> is(3*x^2, Type::Even), is(x + 1, Type::Even)

                           TRUE, FALSE

>> is(x, Type::Integer), is(2*x, Type::Integer),
   is(x/2, Type::Integer), is(x/3, Type::Integer)

                  TRUE, TRUE, TRUE, UNKNOWN

>> assume(y, Type::Odd): is(x + y, Type::Even)

                               FALSE

>> is(2*(x + y), Type::Even)

                               TRUE

>> delete x, y:
```

Changes:

- No changes.

`Type::Imaginary` – a type and a property representing imaginary numbers

`Type::Imaginary` represents complex numbers with vanishing real part. This type can also be used as a property to mark identifiers as imaginary numbers.

Call(s):

- `testtype(obj, Type::Imaginary)`
- `assume(x, Type::Imaginary)`
- `is(ex, Type::Imaginary)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier
`ex` — an arithmetical expression

Parameters:

`obj` — any MuPAD object

Return Value: see `assume`, `is` and `testtype`

Related Functions: `assume`, `is`, `testtype`, `Type::Complex`, `Type::Property`

Details:

- The call `testtype(obj, Type::Imaginary)` checks, whether `obj` is an imaginary number (or zero) and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_COMPLEX` and checks, whether `iszero(Re(obj))` holds, or whether `iszero(obj)` is `TRUE`. This does not include arithmetical expressions such as `I*exp(1)`, which are not identified as of type `Type::Imaginary`.
- The call `assume(x, Type::Imaginary)` marks the identifier `x` as an imaginary number.

  The call `is(ex, Type::Imaginary)` derives, whether the expression `ex` is an imaginary number (or this property can be derived).

- This type represents a property that can be used in `assume` and `is`.
- The call `assume(Re(x) = 0)` has the same meaning as `assume(x, Type::Imaginary)`.

Example 1. The following numbers are of type `Type::Imaginary`:

```
>> testtype(5*I, Type::Imaginary),
   testtype(3/2*I, Type::Imaginary),
   testtype(-1.23*I, Type::Imaginary)

                          TRUE, TRUE, TRUE
```

The following expressions are exact representations of imaginary numbers. However, syntactically they are not of type `Type::Imaginary`, because their domain type is not `DOM_COMPLEX`:

```
>> testtype(exp(3)*I, Type::Imaginary),
   testtype(PI*I, Type::Imaginary),
   testtype(sin(2*I), Type::Imaginary)

                        FALSE, FALSE, FALSE
```

In contrast to `testtype`, the function `is` performs a semantical test:

```
>> is(exp(3)*I, Type::Imaginary),
   is(PI*I, Type::Imaginary),
   is(sin(2*I), Type::Imaginary)

                          TRUE, TRUE, TRUE
```

Example 2. Identifiers may be assumed to represent an imaginary number:

```
>> assume(x, Type::Imaginary): is(x, Type::Imaginary), Re(x), Im(x)

                            TRUE, 0, -I x
```

The imaginary numbers are a subset of the complex numbers:

```
>> is(x, Type::Complex)

                                TRUE

>> unassume(x):
```

Changes:

- No changes.

`Type::IndepOf` – a type representing objects that do not contain given identifiers

`Type::IndepOf(x)` represents objects that do not contain the identifier `x`.

`Type::IndepOf({x1, x2, ...})` represents objects that do not contain any of the identifiers `x1`, `x2` etc.

Call(s):

- `testtype(obj, Type::IndepOf(x))`
- `testtype(obj, Type::IndepOf({x1, ...}))`

Parameters:

`obj` — any MuPAD object
`x, x1, x2` — identifiers of domain type `DOM_IDENT`

Return Value: see `testtype`

Related Functions: `has`, `indets`, `testtype`

Details:

- The call `testtype(obj, Type::IndepOf(x))` checks, whether `obj` does not contain the identifier `x` and returns `TRUE`, if it holds, otherwise `FALSE`.
- `Type::IndepOf` uses `has` to check whether the object contains at least one of the specified identifiers.
- This type expects one argument `x` or `{x1, ...}`.
- This type does not represent a property.

Example 1. The following expression depends on `x`:

```
>> testtype(x^2 - x + 3, Type::IndepOf(x))
```

```
FALSE
```

It is independend of `y`:

```
>> testtype(x^2 - x + 3, Type::IndepOf(y))
```

TRUE

The following expression is independend of `x` and `y`:

```
>> testtype(2*(a + b)/c, Type::IndepOf({x, y}))
```

TRUE

The following call selects all operands of the expression that are independend of `x`:

```
>> select(sin(y) + x^2 - 3*x + 2, testtype, Type::IndepOf(x))
```

sin(y) + 2

Changes:

- `Type::IndepOf` is a new function.

`Type::Integer` – a type and a property representing integers

`Type::Integer` represents integers. This type can also be used as a property to mark identifiers as integers.

Call(s):

- `testtype(obj, Type::Integer)`
- `assume(x, Type::Integer)`
- `is(ex, Type::Integer)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `assume`, `is` and `testtype`

Related Functions: `assume`, `is`, `testtype`, `Type::Real`, `Type::Property`

Details:

- The call `testtype(obj, Type::Integer)` checks, whether `obj` is an integer number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`.
- The call `assume(x, Type::Integer)` marks the identifier `x` as an integer number.

  The call `is(ex, Type::Integer)` derives, whether the expression `ex` is an integer number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::Integer`:

```
>> testtype(0, Type::Integer), testtype(55, Type::Integer),
   testtype(-111, Type::Integer)

                          TRUE, TRUE, TRUE
```

Example 2. We use this type as a property:

```
>> assume(x, Type::Integer):
```

The following calls to `is` derive the properties of a composite expression from the properties of its indeterminates:

```
>> is(3*x, Type::Real), is(2*x, Type::Even), is(x/2, Type::Integer)

                        TRUE, TRUE, UNKNOWN

>> assume(y, Type::Integer): is(x + y^2, Type::Integer)

                                TRUE

>> unassume(x), unassume(y):
```

Changes:

- No changes.

`Type::Interval` – a property representing intervals

`Type::Interval(a, b, ..)` represents the interval (a, b).

`Type::Interval([a], b, ..)` represents the interval $[a, b)$.

`Type::Interval(a, [b], ..)` represents the interval $(a, b]$.

`Type::Interval([a], [b], ..)` represents the interval $[a, b]$.

`Type::Interval([a, b], ..)` represents the interval $[a, b]$.

Call(s):

- `Type::Interval(a, b <, domain>)`
- `Type::Interval([a], b <, domain>)`
- `Type::Interval(a, [b] <, domain>)`
- `Type::Interval([a], [b] <, domain>)`
- `Type::Interval([a, b] <, domain>)`

Parameters:

`a, b` — the borders of the interval: arithmetical objects
`domain` — a type object such as `Type::Real`, `Type::Integer` or `Type::Rational` representing a subset of the real numbers. The default domain is `Type::Real`.

Return Value: a `Type` object

Related Functions: `assume`, `is`, `testtype`, `Type::Integer`, `Type::Rational`, `Type::Real`

Details:

- With the default domain `Type::Real`, the type object created by `Type::Interval` represents a real interval, i.e., the set of all real numbers between the border points `a` and `b`. If another domain is specified, then the type object represents the intersection of the real interval with the set represented by the domain. E.g., `Type::Interval(a, b, Type::Rational)` represents the set of all rational numbers between `a` and `b`.

- The type object represents a property that may be used in `assume` and `is`. With

  ```
  assume(x, Type::Interval(a, b, domain))
  ```

  the identifier `x` is marked as a number from the interval represented by the type object. With

  ```
  is(x, Type::Interval(a, b, domain))
  ```

  one queries, whether `x` is contained in the interval.

- Interval types should not be used in `testtype`. No MuPAD object matches these types syntactically, i.e., `testtype` always returns `FALSE`.

Example 1. The following type object represents the open interval $(-1, 1)$:

```
>> Type::Interval(-1, 1)

                        ]-1, 1[ of Type::Real
```

The following calls are equivalent: both create the type representing a closed interval:

```
>> Type::Interval([-1], [1]), Type::Interval([-1, 1])

           [-1, 1] of Type::Real, [-1, 1] of Type::Real
```

The following call creates the type representing the set of all integers from -10 to 10:

```
>> Type::Interval([-10, 10], Type::Integer)

                    [-10, 10] of Type::Integer
```

The following call creates the type representing the set of all rational numbers in the interval $[0, 1)$:

```
>> Type::Interval([0], 1, Type::Rational)

                      [0, 1[ of Type::Rational
```

Example 2. We use intervals as a property. The following call marks x as a real number from the interval $[0, 2)$:

```
>> assume(x, Type::Interval([0], 2)):
```

Consequently, $x^2 + 1$ lies in the interval $[1, 5)$:

```
>> is(x^2 + 1 >= 1), is(x^2 + 1 < 5)
```

```
                                TRUE, TRUE
```

The following call marks x as an integer larger than -10 and smaller than 100:

```
>> assume(x, Type::Interval(-10, 100, Type::Integer)):
```

Consequently, x^3 is an integer larger than -730 and smaller than 970300:

```
>> is(x^3, Type::Integer), is(x^3 >= -729), is(x^3 < 970300),
   is(x^3, Type::Interval(-10^3, 100^3, Type::Integer))

                    TRUE, TRUE, TRUE, TRUE

>> is(x <= -730), is(x^3 >= 970300)

                          FALSE, FALSE

>> is(x > 0), is(x^3, Type::Interval(0, 10, Type::Integer))

                        UNKNOWN, UNKNOWN

>> unassume(x):
```

Changes:

- the internal structure was revised completely
- the output is changed

`Type::ListOf` – type for testing lists of objects with the same type

`Type::ListOf` describes lists of objects of a specified type.

Call(s):

- `testtype(obj, Type::ListOf(obj_type <, min_nr <, max_nr>>))`

Parameters:

`obj`	—	any MuPAD object
`obj_type`	—	the type of the objects; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`
`min_nr`	—	the minimal number of objects as nonnegative integer
`max_nr`	—	the maximal number of objects as nonnegative integer

Return Value: see `testtype`

Related Functions: `DOM_LIST`, `testtype`, `Type::ListProduct`, `Type::SetOf`, `Type::Union`

Details:

- The call `testtype(obj, Type::ListOf(obj_types, ...))` checks, whether `obj` is a list with elements of the given type `obj_type, ...` and returns `TRUE`, if it holds, otherwise `FALSE`.
- This type expects one or more arguments `obj_type, ...,<, min_nr <, max_nr>>`.
- The two optional parameters `min_nr` and `max_nr` determine the minimum and maximum number of elements in the analyzed list. If the numbers are not be given, the number of elements in the list will not be checked. If only the minimum is given, only the minimal number of elements in the list is checked.
- Note especially that `Type::Union` provides a way to allow more than one type for the list elements.
- This type does not represent a property.

Example 1. Is the given list a list of identifiers?

```
>> testtype([a, b, c, d, e, f], Type::ListOf(DOM_IDENT))

                                TRUE
```

Is the given list a list of at least five real numbers?

```
>> testtype([0, 0.5, 1, 1.5, 2, 2.5, 3], Type::ListOf(Type::Real, 5))

                                TRUE
```

Example 2. `testtype` is used to select lists with exactly two identifiers:

```
>> S := {[a], [a, b], [d, 1], [0, d], [e], [d, e]}:
   select(S, testtype, Type::ListOf(DOM_IDENT, 2, 2))

                          {[a, b], [d, e]}
```

Changes:

- No changes.

`Type::ListProduct` – type for testing lists

With `Type::ListProduct`, lists with different object types can be identified.

Call(s):

- `testtype(obj, Type::ListProduct(typedef, ...))`

Parameters:

`obj`	—	any MuPAD object
`typedef`	—	a sequence of types; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

Return Value: see `testtype`

Related Functions: `testtype`, `Type::ListOf`, `Type::Product`

Details:

- The call `testtype(obj, Type::ListProduct(typedef))` checks, whether `obj` is a list of objects, which have the types given by `typedef` and returns `TRUE`, if it holds, otherwise `FALSE`.
- `obj` must have the same number of arguments as the sequence `typedef`. The elements of `obj` are checked one after another: the first element of `obj` is checked against the type given by the first element of `typedef`, and so on. All elements and types must match.
- This type expects one or more arguments `Type::ListProduct(typedef, ...)`.
- `typedef, ...` must be a nonempty sequence of types. A type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`.
- This type does not represent a property.

Example 1. The argument is a list of a positive integer followed by an identifier:

```
>> testtype([5, x], Type::ListProduct(Type::PosInt, Type::Unknown))
```

```
TRUE
```

Is the argument is a list of a five positive integers? (We use `$` here to repeat `Type::PosInt` five times.)

```
>> testtype([5, 3, 5, -1, 0], Type::ListProduct(Type::PosInt $ 5))
```

```
FALSE
```

Changes:

- No changes.

`Type::NegInt` – a type and a property representing negative integers

`Type::NegInt` represents negative integers. `Type::NegInt` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::NegInt)`
- `assume(x, Type::NegInt)`
- `is(ex, Type::NegInt)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Property`

Details:

- The call `testtype(obj, Type::NegInt)` checks, whether `obj` is a negative integer number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(obj < 0)` holds.
- The call `assume(x, Type::NegInt)` marks the identifier `x` as a negative integer number.

  The call `is(ex, Type::NegInt)` derives, whether the expression `ex` is a negative integer number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::NegInt`:

```
>> testtype(-2, Type::NegInt),
   testtype(-3, Type::NegInt),
   testtype(-55, Type::NegInt),
   testtype(-1, Type::NegInt),
   testtype(-111111111, Type::NegInt)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is a negative integer:

```
>> assume(x, Type::NegInt):
   is(x, Type::NegInt)

                              TRUE
```

Negative integers are integers, of course:

```
>> assume(x, Type::NegInt):
   is(x, Type::Integer)

                              TRUE
```

However, integers can be negative or not:

```
>> assume(x, Type::Integer):
   is(x, Type::NegInt)

                            UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::NegRat` – a type and a property representing negative rational numbers

`Type::NegRat` represents negative rational numbers. `Type::NegRat` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::NegRat)`
- `assume(x, Type::NegRat)`
- `is(ex, Type::NegRat)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Property`

Details:

- The call `testtype(obj, Type::NegRat)` checks, whether `obj` is a negative rational number and returns TRUE, if it holds, otherwise FALSE.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and `DOM_RAT` and checks, if `bool(obj < 0)` holds.
- The call `assume(x, Type::NegRat)` marks the identifier `x` as a negative rational number.

  The call `is(ex, Type::NegRat)` derives, whether the expression `ex` is a negative rational number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::NegRat`:

```
>> testtype(-2, Type::NegRat),
   testtype(-3/4, Type::NegRat),
   testtype(-55/111, Type::NegRat),
   testtype(-1, Type::NegRat),
   testtype(-111/111111, Type::NegRat)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is negative rational:

```
>> assume(x, Type::NegRat):
   is(x, Type::NegRat)

                              TRUE
```

Also negative rational numbers are rational:

```
>> assume(x, Type::NegRat):
   is(x, Type::Rational)

                              TRUE
```

However, rational numbers can be negative rational or not:

```
>> assume(x, Type::Rational):
   is(x, Type::NegRat)

                            UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::Negative` – a type and a property representing negative numbers

`Type::Negative` represents negative numbers. `Type::Negative` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::Negative)`
- `assume(x, Type::Negative)`
- `is(ex, Type::Negative)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Real`, `Type::Property`

Details:

- The call `testtype(obj, Type::Negative)` checks, whether `obj` is a negative real number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT` and `DOM_FLOAT` and checks, if `bool(obj < 0)` holds. This does not include arithmetical expressions such as `-exp(1)`, which are not identified as of type `Type::Negative`.
- The call `assume(x, Type::Negative)` marks the identifier `x` as a negative real number.

  The call `is(ex, Type::Negative)` derives, whether the expression `ex` is a negative real number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.
- Instead of `Type::Negative` the assumption can also be `assume(x < 0)`.

Example 1. The following numbers are of type `Type::Negative`:

```
>> testtype(-2, Type::Negative),
   testtype(-3/4, Type::Negative),
   testtype(-0.123, Type::Negative),
   testtype(-1, Type::Negative),
   testtype(-1.02, Type::Negative)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of negative numbers, but syntactically they are not of `Type::Negative`:

```
>> testtype(-exp(1), Type::Negative),
   testtype(-PI^2 - 5, Type::Negative),
   testtype(-sin(2), Type::Negative)

                        FALSE, FALSE, FALSE
```

Example 2. Assume an identifier is negative:

```
>> assume(x, Type::Negative):
   is(x, Type::Negative)

                                  TRUE
```

This is equal to:

```
>> assume(x < 0):
   is(x < 0)

                                  TRUE
```

Also negative numbers are real:

```
>> assume(x, Type::Negative):
   is(x, Type::Real)

                                  TRUE
```

However, real numbers can be negative or not:

```
>> assume(x, Type::Real):
   is(x, Type::Negative)

                                UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::NonNegInt` – a type and a property representing nonnegative integers

`Type::NonNegInt` represents nonnegative integers. `Type::NonNegInt` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::NonNegInt)`
- `assume(x, Type::NonNegInt)`
- `is(ex, Type::NonNegInt)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Integer`, `Type::Property`

Details:

- The call `testtype(obj, Type::NonNegInt)` checks, whether `obj` is a nonnegative integer number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(obj >= 0)` holds.
- The call `assume(x, Type::NonNegInt)` marks the identifier `x` as a nonnegative integer number.

  The call `is(ex, Type::NonNegInt)` derives, whether the expression `ex` is a nonnegative integer number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::NonNegInt`:

```
>> testtype(2, Type::NonNegInt),
   testtype(3/4, Type::NonNegInt),
   testtype(55/111, Type::NonNegInt),
   testtype(1, Type::NonNegInt),
   testtype(111/111111, Type::NonNegInt)

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

Example 2. Assume an identifier is nonnegative rational:

```
>> assume(x, Type::NonNegInt):
   is(x, Type::NonNegInt)

                                TRUE
```

Also nonnegative integers are integers:

```
>> assume(x, Type::NonNegInt):
   is(x, Type::Integer)

                                TRUE
```

However, integers can be nonnegative or not:

```
>> assume(x, Type::Integer):
   is(x, Type::NonNegInt)

                              UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::NonNegRat` – a type and a property representing nonnegative rational numbers

`Type::NonNegRat` represents nonnegative rational numbers. `Type::NonNegRat` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::NonNegRat)`
- `assume(x, Type::NonNegRat)`
- `is(ex, Type::NonNegRat)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Rational`, `Type::Property`

Details:

- The call `testtype(obj, Type::NonNegRat)` checks, whether `obj` is a nonnegative rational number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and `DOM_RAT` and checks, if `bool(obj >= 0)` holds.
- The call `assume(x, Type::NonNegRat)` marks the identifier `x` as a nonnegative rational number.

  The call `is(ex, Type::NonNegRat)` derives, whether the expression `ex` is a nonnegative rational number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::NonNegRat`:

```
>> testtype(2, Type::NonNegRat),
   testtype(3/4, Type::NonNegRat),
   testtype(55/111, Type::NonNegRat),
   testtype(0, Type::NonNegRat),
   testtype(111/111111, Type::NonNegRat)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is nonnegative rational:

```
>> assume(x, Type::NonNegRat):
   is(x, Type::NonNegRat)

                              TRUE
```

Also nonnegative rational numbers are rational:

```
>> assume(x, Type::NonNegRat):
   is(x, Type::Rational)

                              TRUE
```

However, rational numbers can be nonnegative rational or not:

```
>> assume(x, Type::Rational):
   is(x, Type::NonNegRat)

                                UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::NonNegative` – a type and a property representing nonnegative numbers

`Type::NonNegative` represents nonnegative numbers. `Type::NonNegative` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::NonNegative)`
- `assume(x, Type::NonNegative)`
- `is(ex, Type::NonNegative)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Real`, `Type::Property`

Details:

- The call `testtype(obj, Type::NonNegative)` checks, whether `obj` is a nonnegative real number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT` and `DOM_FLOAT` and checks, if `bool(obj >= 0)` holds. This does not include arithmetical expressions such as `exp(1)`, which are not identified as of type `Type::NonNegative`.

- The call `assume(x, Type::NonNegative)` marks the identifier `x` as a nonnegative real number.

  The call `is(ex, Type::NonNegative)` derives, whether the expression `ex` is a nonnegative real number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.
- Instead of `Type::NonNegative` the assumption can also be `assume(x >= 0)`.

Example 1. The following numbers are of type `Type::NonNegative`:

```
>> testtype(2, Type::NonNegative),
   testtype(3/4, Type::NonNegative),
   testtype(0.123, Type::NonNegative),
   testtype(0, Type::NonNegative),
   testtype(1.02, Type::NonNegative)

                  TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of nonnegative numbers, but syntactically they are not of `Type::NonNegative`:

```
>> testtype(exp(1), Type::NonNegative),
   testtype(PI^2 + 5, Type::NonNegative),
   testtype(sin(2), Type::NonNegative)

                      FALSE, FALSE, FALSE
```

The function `is`, however, can find these expressions to be nonnegative:

```
>> is(exp(1), Type::NonNegative),
   is(PI^2 + 5, Type::NonNegative),
   is(sin(2), Type::NonNegative)

                       TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is nonnegative:

```
>> assume(x, Type::NonNegative):
   is(x, Type::NonNegative)

                              TRUE
```

This is equal to:

```
>> assume(x >= 0):
   is(x >= 0)
```

```
                                TRUE
```

Also nonnegative numbers are real:

```
>> assume(x, Type::NonNegative):
   is(x, Type::Real)

                                TRUE
```

But real numbers can be nonnegative or not:

```
>> assume(x, Type::Real):
   is(x, Type::NonNegative)

                              UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::NonZero` – a type and a property representing "unequal to zero"

`Type::NonZero` is a type of objects unequal to zero. `Type::NonZero` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::NonZero)`
- `assume(x, Type::NonZero)`
- `is(ex, Type::NonZero)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Zero`

Details:

- The call `testtype(obj, Type::NonZero)` checks, whether `obj` is *not* zero and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test and uses the function `iszero` to determine, whether the object is not zero. This implies that identifiers without a value, for example, are considered as being different from zero, see example 1.
- The call `assume(x, Type::NonZero)` marks the identifier `x` as a complex number unequal to zero.

  The call `is(ex, Type::NonZero)` derives, whether the expression `ex` is a complex number unequal to zero (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.
- The call `assume(x <> 0)` has the same meaning as `assume(x, Type::NonZero)`.

Example 1. Usage of `Type::NonZero` whith `testtype`:

```
>> testtype(1.0, Type::NonZero)
```

```
TRUE
```

Since `iszero(x)` returns `FALSE`, the following call returns `TRUE`:

```
>> testtype(x, Type::NonZero)
```

```
TRUE
```

Example 2. Usage of `Type::NonZero` whith `assume` and `is`:

```
>> is(x, Type::NonZero)
```

```
UNKNOWN
```

Assumption: `x` is `Type::NonZero`:

```
>> assume(x, Type::NonZero):
   is(x, Type::NonZero)
```

```
TRUE
```

The same again:

```
>> assume(x <> 0):
   is(x <> 0)
```

```
                                TRUE
```

The difference between `testtype` and `is`:

```
>> delete x:
   is(x, Type::NonZero), testtype(x, Type::NonZero)

                            UNKNOWN, TRUE
```

`x` could be zero:

```
>> assume(x >= 0):
   is(x, Type::NonZero), testtype(x, Type::NonZero)

                            UNKNOWN, TRUE

>> delete x:
```

Changes:

- No changes.

`Type::Numeric` – a type for testing numerical objects

With `Type::Numeric`, numeric objects (numbers) can be identified.

Call(s):

- `testtype(obj, Type::Numeric)`

Parameters:

`obj` — any MuPAD object

Return Value: see `testtype`

Related Functions: `assume`, `is`, `testtype`, `Type::Complex`

Details:

- The call `testtype(obj, Type::Numeric)` checks, whether `obj` is a number and returns `TRUE`, if it holds, otherwise `FALSE`.
- A number has the domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_COMPLEX`.
- This type does not represent a property.

Example 1. The following objects are numbers.

```
>> testtype(2, Type::Numeric),
   testtype(3/4, Type::Numeric),
   testtype(0.123, Type::Numeric),
   testtype(1 + I/3, Type::Numeric),
   testtype(1.0 + 2.0*I, Type::Numeric)

                  TRUE, TRUE, TRUE, TRUE, TRUE
```

The following objects are not numerical objects.

```
>> testtype(ln(2), Type::Numeric),
   testtype(sin(3/4), Type::Numeric),
   testtype(x + I/3, Type::Numeric)

                       FALSE, FALSE, FALSE
```

Changes:

- No changes.

`Type::Odd` – a type and a property representing odd integers

`Type::Odd` represents odd integers. `Type::Odd` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::Odd)`
- `assume(x, Type::Odd)`
- `is(ex, Type::Odd)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Even`, `Type::Property`

Details:

- The call `testtype(obj, Type::Odd)` checks, whether `obj` is an odd number and returns TRUE, if it holds, otherwise FALSE.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(domtype((x-1)/2) = DOM_INT)` holds.
- The call `assume(x, Type::Odd)` marks the identifier `x` as an odd number.

  The call `is(ex, Type::Odd)` derives, whether the expression `ex` is an odd number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::Odd`:

```
>> testtype(1, Type::Odd),
   testtype(-3, Type::Odd),
   testtype(7, Type::Odd),
   testtype(-11113, Type::Odd),
   testtype(4185296581467695597, Type::Odd)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is odd:

```
>> assume(x, Type::Odd):
   is(x, Type::Odd)

                              TRUE
```

All odd numbers are integer:

```
>> assume(x, Type::Odd):
   is(x, Type::Integer)

                              TRUE
```

However, integers can be odd or not:

```
>> assume(x, Type::Integer):
   is(x, Type::Odd)

                            UNKNOWN
```

However, even numbers are not odd:

```
>> assume(x, Type::Odd):
   is(2*x, Type::Odd)

                                FALSE

>> assume(n, Type::Even):
   is(x*n, Type::Odd)

                                FALSE

>> is(x*n + 1, Type::Odd)

                                TRUE

>> delete x, n:
```

Changes:

- No changes.

`Type::PolyExpr` – type for testing polynomial expressions

With `Type::PolyExpr`, polynomial expressions can be identified.

Call(s):

- `testtype(obj, Type::PolyExpr(unknowns <, coeff_type>))`

Parameters:

`obj`	—	any MuPAD object
`unknowns`	—	an indeterminate or a list of indeterminates
`coeff_type`	—	the type of the coefficients; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

Return Value: see `testtype`

Related Functions: `testtype`, `Type::PolyOf`, `poly`, `indets`

Details:

- The call `testtype(obj, Type::PolyExpr(unknowns))` checks, whether `obj` is a polynomial expression in the indeterminates `unknowns` and, if so, returns `TRUE`, otherwise `FALSE`.
- A polynomial expression in `indet` is a MuPAD expression, where `indet` occurs only as operand of `_plus` or `_mult` expressions and in the base of `_power` with a positive integer exponent.
- A polynomial expression is a representation of a polynomial, but it has the MuPAD type `DOM_EXPR` and is not produced by the function `poly`.
- This type expects one or more arguments `Type::PolyExpr(indets <, coeff_type>)`.

  `indets` must be an identifier or a list of identifiers.

  The optional argument `coeff_type` determines the type of the coefficients. If it is not given, `Type::AnyType` will be used.
- This type does not represent a property.

Example 1. Is the object a polynomial expression with variable `x`?

```
>> X := -x^2 - x + 3:
   testtype(X, Type::PolyExpr(x))
```

```
                                TRUE
```

But `X` is not a MuPAD polynomial in `x`:

```
>> testtype(X, Type::PolyOf(x))
```

```
                                FALSE
```

Is the object a polynomial expression with variables `x` and `y` and with integer coefficients?

```
>> X := -x^2 - x + 3:
   testtype(X, Type::PolyExpr([x, y], Type::Integer))
```

```
                                TRUE
```

The next example too?

```
>> X := -x^2 - y^2 + 3*x + 3*y - 1:
   testtype(X, Type::PolyExpr([x, y], Type::Integer))
```

```
                                TRUE
```

```
>> delete X:
```

Changes:

- No changes.

`Type::PolyOf` – type for testing polynomials

With `Type::PolyOf`, polynomials can be identified.

Call(s):

- `testtype(obj, Type::PolyOf(coeff_type <, num_ind>))`

Parameters:

`obj`	—	any MuPAD object
`coeff_type`	—	the type of the coefficientes; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`
`num_ind`	—	the number of indeterminates

Return Value: see `testtype`

Related Functions: `testtype`, `poly`, `indets`

Details:

- The call `testtype(obj, Type::PolyOf(coeff_type))` checks, whether `obj` is a polynomial with coefficients of type `coeff_type` and, if so, returns `TRUE`, otherwise `FALSE`.

- Only polynomials of type `DOM_POLY` can be identified with `Type::PolyOf`, see `Type::PolyExpr` for polynomial expressions.

- This type expects one or more arguments `Type::PolyOf(coeff_type <, num_ind>)`.

  `coeff_type` determines the type of the coefficients.

  The optional argument `num_ind` determines the number of indeterminates. If this argument is not given, the polynomial may have any number of indeterminates.

- This type does not represent a property.

Example 1. Is the object a polynomial with integer coefficients?

```
>> P := poly(-x^2 - x + 3):
   testtype(P, Type::PolyOf(Type::Integer))
```

```
                                TRUE
```

Is the object a polynomial with integer coefficients and two indets?

```
>> P := poly(-x^2 - x + 3, [x, y]):
   testtype(P, Type::PolyOf(Type::Integer, 2))
```

```
                                TRUE
```

```
>> delete P:
```

Changes:

- No changes.

`Type::PosInt` – a type and a property representing positive integers

`Type::PosInt` represents positive integers. `Type::PosInt` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::PosInt)`
- `assume(x, Type::PosInt)`
- `is(ex, Type::PosInt)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Property`

Details:

- The call `testtype(obj, Type::PosInt)` checks, whether `obj` is a positive integer number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `bool(obj > 0)` holds.
- The call `assume(x, Type::PosInt)` marks the identifier `x` as a positive integer number.

  The call `is(ex, Type::PosInt)` derives, whether the expression `ex` is a positive integer number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::PosInt`:

```
>> testtype(2, Type::PosInt),
   testtype(3, Type::PosInt),
   testtype(55, Type::PosInt),
   testtype(1, Type::PosInt),
   testtype(111, Type::PosInt)
```

```
                TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is positive integer:

```
>> assume(x, Type::PosInt):
   is(x, Type::PosInt)
```

```
                              TRUE
```

Also positive integers are integers:

```
>> assume(x, Type::PosInt):
   is(x, Type::Integer)
```

```
                              TRUE
```

However, integers can be positive or not:

```
>> assume(x, Type::Integer):
   is(x, Type::PosInt)
```

```
                            UNKNOWN
```

```
>> delete x:
```

Changes:

- No changes.

`Type::PosRat` – a type and a property representing positive rational numbers

`Type::PosRat` represents positive rational numbers. `Type::PosRat` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::PosRat)`
- `assume(x, Type::PosRat)`
- `is(ex, Type::PosRat)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Property`

Details:

- The call `testtype(obj, Type::PosRat)` checks, whether `obj` is a positive rational number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and `DOM_RAT` and checks, if `bool(obj > 0)` holds.
- The call `assume(x, Type::PosRat)` marks the identifier `x` as a positive rational number.

  The call `is(ex, Type::PosRat)` derives, whether the expression `ex` is a positive rational number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::PosRat`:

```
>> testtype(2, Type::PosRat),
   testtype(3/4, Type::PosRat),
   testtype(55/111, Type::PosRat),
   testtype(1, Type::PosRat),
   testtype(111/111111, Type::PosRat)

                  TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is positive rational:

```
>> assume(x, Type::PosRat):
   is(x, Type::PosRat)

                                TRUE
```

Also positive rational numbers are rational:

```
>> assume(x, Type::PosRat):
   is(x, Type::Rational)

                                TRUE
```

However, rational numbers can be positive rational or not:

```
>> assume(x, Type::Rational):
   is(x, Type::PosRat)

                              UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::Positive` – a type and a property representing positive numbers

`Type::Positive` represents positive numbers. `Type::Positive` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::Positive)`
- `assume(x, Type::Positive)`
- `is(ex, Type::Positive)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Property`

Details:

- The call `testtype(obj, Type::Positive)` checks, whether `obj` is a positive real number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT` and `DOM_FLOAT` and checks, if `bool(obj > 0)` holds. This does not include arithmetical expressions such as `exp(1)`, which are not identified as of type `Type::Positive`.
- The call `assume(x, Type::Positive)` marks the identifier `x` as a positive real number.

  The call `is(ex, Type::Positive)` derives, whether the expression `ex` is a positive real number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.
- Instead of `Type::Positive` the assumption can also be `assume(x > 0)`.

Example 1. The following numbers are of type `Type::Positive`:

```
>> testtype(2, Type::Positive),
   testtype(3/4, Type::Positive),
   testtype(0.123, Type::Positive),
   testtype(1, Type::Positive),
   testtype(1.02, Type::Positive)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of positive numbers, but syntactically they are not of `Type::Positive`:

```
>> testtype(exp(1), Type::Positive),
   testtype(PI^2 + 5, Type::Positive),
   testtype(sin(2), Type::Positive)

                        FALSE, FALSE, FALSE
```

Ths function `is`, however, realizes that they are, indeed, positive:

```
>> is(exp(1), Type::Positive),
   is(PI^2 + 5, Type::Positive),
   is(sin(2), Type::Positive)

                         TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is positive:

```
>> assume(x, Type::Positive):
   is(x, Type::Positive)

                                TRUE
```

This is equivalent to:

```
>> assume(x > 0):
   is(x > 0)

                                TRUE
```

Also positive numbers are real:

```
>> assume(x, Type::Positive):
   is(x, Type::Real)

                                TRUE
```

But real numbers can be positive or not:

```
>> assume(x, Type::Real):
   is(x, Type::Positive)

                               UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::Prime` – a type and a property representing prime numbers

`Type::Prime` represents prime numbers. `Type::Prime` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::Prime)`
- `assume(x, Type::Prime)`
- `is(ex, Type::Prime)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `isprime`, `Type::Property`

Details:

- The call `testtype(obj, Type::Prime)` checks, whether `obj` is a prime number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and checks, if `isprime(obj)` holds.
- The call `assume(x, Type::Prime)` marks the identifier `x` as a prime number.

  The call `is(ex, Type::Prime)` derives, whether the expression `ex` is a prime number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::Prime`:

```
>> testtype(2, Type::Prime),
   testtype(3, Type::Prime),
   testtype(7, Type::Prime),
   testtype(11113, Type::Prime),
   testtype(4185296581467695597, Type::Prime)

                  TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. Assume an identifier is prime:

```
>> assume(x, Type::Prime):
   is(x, Type::Prime)

                                TRUE
```

Also prime numbers are integers:

```
>> assume(x, Type::Prime):
   is(x, Type::Integer)

                                TRUE
```

However, integer numbers can be prime or not:

```
>> assume(x, Type::Integer):
   is(x, Type::Prime)

                              UNKNOWN

>> delete x:
```

Changes:

- No changes.

`Type::Product` – type for testing sequences

`Type::Product` is the type of sequences of objects of different types.

Call(s):

- `testtype(obj, Type::Product(typedef, ...))`

Parameters:

`obj` — any MuPAD object
`typedef` — a sequence of types; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

Return Value: see `testtype`

Related Functions: `testtype`, `Type::ListProduct`

Details:

- The call `testtype(obj, Type::Product(typedef))` checks, whether `obj` is a sequence of objects, which have the types given by `typedef` and returns `TRUE`, if it holds, otherwise `FALSE`.
- `obj` must have the same number of arguments as the sequence `typedef`. The elements of `obj` are checked one after another: the first element of `obj` is checked against the type given by the first element of `typedef` and so on. All elements and types must match.
- This type expects one or more arguments `Type::Product(typedef, ...)`.
- `typedef, ...` must be a nonempty sequence of types. A type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`.
- This type does not represent a property.

Example 1. The argument is a sequence of a positive integer followed by an identifier:

```
>> testtype((5, x), Type::Product(Type::PosInt, Type::Unknown))
```

```
TRUE
```

Is the argument is a sequence of five positive integers? (For help on `$` see `_seqgen`.)

```
>> testtype((5, 3, 5, -1, 0), Type::Product(Type::PosInt $ 5))
```

```
FALSE
```

Changes:

- No changes.

`Type::Property` – type to identify properties

With `Type::Property`, properties can be identified.

Call(s):

- `testtype(obj, Type::Property)`

Parameters:

`obj` — any MuPAD object

Return Value: see `testtype`

Related Functions: `testtype`, `is`

Details:

- The call `testtype(obj, Type::Property)` checks, whether the MuPAD object `obj` is a property and returns `TRUE`, if it holds, otherwise `FALSE`.
- Some elements of the library `Type` serve two functions. One is to perform syntactical tests to identify the type of an object (with `testtype`), the other is to occur as a property within `assume` and `is`.

  `Type::Property` itself is not a property.

- To determine whether an element of `Type` is a property, `Type::Property` can be used with `testtype`.
- This type does not represent a property.

Example 1. Is `Type::PosInt` a property?

```
>> testtype(Type::PosInt, Type::Property)
```

TRUE

Also an interval created with `Type::Interval` is a property:

```
>> testtype(Type::Interval(0, 1), Type::Property)
```

```
                                TRUE
```

Is `Type::Constant` a property?

```
>> testtype(Type::Constant, Type::Property)
```

```
                               FALSE
```

`Type::Constant` is not a property and cannot be used as argument of `assume`:

```
>> assume(x, Type::Constant)
```

```
 Error: second argument must be a property [property::assume]
```

The next example shows the usage of `testtype` to select properties among operands of `Type`:

```
>> T := Type::Numeric, Type::PosInt, Type::Unknown, Type::Zero:
   select(T, testtype, Type::Property)
```

```
                    Type::PosInt, Type::Zero
```

```
>> delete x, T:
```

Changes:

- `Type::Property` is a new function.

`Type::RatExpr` – type for testing rational expressions

With `Type::RatExpr`, rational expressions can be identified.

Call(s):

- `testtype(obj, Type::RatExpr(indet <, coeff_type>))`

Parameters:

`obj`	—	any MuPAD object
`indet`	—	an indeterminante
`coeff_type`	—	a type for the coefficientes; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

Return Value: see `testtype`

Related Functions: `testtype`, `indets`

Details:

- The call `testtype(obj, Type::RatExpr(indet))` checks, whether `obj` is a rational expression in the indeterminante `indet`, i.e., the quotient of two polynomial expressions in `indet`. If it is, the result is `TRUE`, otherwise `FALSE`.
- A rational expression in `indet` is a MuPAD expression, and `indet` occurs only as operand of `_plus` or `_mult` expressions and in `_power` with an integer exponent.
- This type expects one or more arguments `Type::RatExpr(indet <, coeff_type>)`.
- `indet` must be an identifier, and `coeff_type` a type for the coefficients of the rational expression.
- This type does not represent a property.

Example 1. A polynomial expression in `x` is also a rational expression in `x`:

```
>> testtype(-x^2 - x + 3, Type::RatExpr(x))
```

```
                                TRUE
```

`testtype` is used to select all rational operands in `x` with positive integer coefficients:

```
>> EX := sin(x) + x^2 - 3*x + 2 + 3/x:
   select(EX, testtype, Type::RatExpr(x, Type::PosInt))
```

```
                          3    2
                          - + x  + 2
                          x
```

```
>> delete EX:
```

Changes:

- No changes.

`Type::Rational` – a type and a property representing rational numbers

`Type::Rational` represents rational numbers. `Type::Rational` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::Rational)`
- `assume(x, Type::Rational)`
- `is(ex, Type::Rational)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Property`

Details:

- The call `testtype(obj, Type::Rational)` checks, whether `obj` is a rational number and returns `TRUE`, if it holds, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT` and `DOM_RAT`.
- The call `assume(x, Type::Rational)` marks the identifier `x` as a rational number.

  The call `is(ex, Type::Rational)` derives, whether the expression `ex` is a rational number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::Rational`:

```
>> testtype(2, Type::Rational),
   testtype(3/4, Type::Rational),
   testtype(-1/2, Type::Rational),
   testtype(-1, Type::Rational),
   testtype(1024/11111, Type::Rational)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

Example 2. Integers are rational:

```
>> assume(x, Type::Integer):
   is(x, Type::Rational)
```

```
                                TRUE
```

However, rational numbers can be integer or not:

```
>> assume(x, Type::Rational):
   is(x, Type::Integer)
```

```
                              UNKNOWN
```

```
>> delete x:
```

Changes:

- No changes.

`Type::Real` – a type and a property representing real numbers

`Type::Real` represents real numbers. `Type::Real` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::Real)`
- `assume(x, Type::Real)`
- `is(ex, Type::Real)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::Property`

Details:

- The call `testtype(obj, Type::Real)` checks, whether `obj` is a real number and, if it is, returns `TRUE`, otherwise `FALSE`.
- `testtype` only performs a syntactical test identifying MuPAD objects of type `DOM_INT`, `DOM_RAT` and `DOM_FLOAT`. This does not include arithmetical expressions such as `exp(1)`, which are not identified as of type `Type::Real`.
- The call `assume(x, Type::Real)` marks the identifier `x` as a real number.

  The call `is(ex, Type::Real)` derives, whether the expression `ex` is a real number (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.

Example 1. The following numbers are of type `Type::Real`:

```
>> testtype(2, Type::Real),
   testtype(3/4, Type::Real),
   testtype(0.123, Type::Real),
   testtype(-1, Type::Real),
   testtype(-1.02, Type::Real)

                TRUE, TRUE, TRUE, TRUE, TRUE
```

The following expressions are exact representations of real numbers, but syntactically they are not of `Type::Real`:

```
>> testtype(exp(1), Type::Real),
   testtype(PI^2 + 5, Type::Real),
   testtype(sin(2), Type::Real)

                      FALSE, FALSE, FALSE
```

The function `is` performs a semantical, mathematically more useful check:

```
>> is(exp(1), Type::Real),
   is(PI^2 + 5, Type::Real),
   is(sin(2), Type::Real)

                       TRUE, TRUE, TRUE
```

Example 2. Integers are real numbers:

```
>> assume(x, Type::Integer):
   is(x, Type::Real)
```

```
                                TRUE
```

But real numbers can be integer or not:

```
>> assume(x, Type::Real):
   is(x, Type::Integer)
```

```
                              UNKNOWN
```

The sine of a real number is a real number in the interval $[-1, 1]$:

```
>> getprop(sin(x))
```

```
                      [-1, 1] of Type::Real
```

```
>> delete x:
```

Changes:

- No changes.

`Type::Relation` – type for testing relations

With `Type::Relation`, relational expression can be identified.

Call(s):

- `testtype(obj, Type::Relation)`

Parameters:

`obj` — any MuPAD object

Return Value: see `testtype`

Related Functions: `testtype`

Details:

- The call `testtype(obj, Type::Relation)` checks, whether `obj` is a relational expression and returns `TRUE`, if it is, otherwise `FALSE`.
- A relation in MuPAD is an expression of the type `"_equal"`, `"_unequal"`, `"_less"` and `"_leequal"`.

  Expressions with the operations `>=` and `>` will be interpreted as expressions with `<=` and `<` by exchanging the operands (see example 2).

- This type does not represent a property.

Example 1. `x > 3` is a relation, while `TRUE` is not:

```
>> testtype(x > 3, Type::Relation),
   testtype(TRUE, Type::Relation)

                              TRUE, FALSE
```

Example 2. MuPAD always interprets expressions with the operations `>=` and `>` as expressions with `<=` and `<` with the operands exchanged:

```
>> x > 3;
   prog::exprtree(x > 3):

                                 3 < x

                                 _less
                                 |
                                 +-- 3
                                 |
                                 '-- x
```

The operator is *not* `>`, but `<`, and the operands have been swapped:

```
>> op(x > 3, 0..2)

                              _less, 3, x
```

Changes:

- No changes.

`Type::Residue` – a property representing a residue class

`Type::Residue(rem, class)` represents the integers n for which $n - rem$ is divisible by *class*.

Call(s):

- `assume(x, Type::Residue(rem, class <, subset>))`
- `is(ex, Type::Residue(rem, class <, subset>))`
- `testtype(obj, Type::Residue(rem, class <, subset>))`

Parameters:

`x`	—	an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`rem`	—	remainder as integer number between `0` and `class - 1`; an integer larger than `class - 1` will be divided by `class` and `rem` gets the remainder of this division
`class`	—	the divider as positive integer
`subset`	—	a subset of the integers (e.g., `Type::PosInt`); otherwise `Type::Integer` is used
`ex`	—	an arithmetical expression
`obj`	—	any MuPAD object

Return Value: see `assume`, `is` and `testtype`

Related Functions: `assume`, `is`, `testtype`, `Type::Even`, `Type::Integer`, `Type::Odd`

Details:

- The call `assume(x, Type::Residue(rem, class))` marks the identifier `x` as an integer divisible by `class` with remainder `rem`.

  The call `is(ex, Type::Residue(rem, class))` derives, whether the expression `ex` is an integer divisible by `class` with remainder `rem` (or this property can be derived).

- This type expects two or three arguments `rem`, `class`, <`subset`>.

- This type represents a property that can be used in `assume` and `is`.

- `Type::Even` and `Type::Odd` are objects created by `Type::Residue`.

- The call `testtype(obj, Type::Residue(rem, class))` checks, whether `obj` is an integer and is divisible by `class` with remainder `rem`. If the optional argument `subset` is given, `testtype` checks additionally `testtype(obj, subset)`.

Example 1. `Type::Residue` can be used in `testtype`:

```
>> testtype(6, Type::Residue(2, 4)),
   testtype(13, Type::Residue(1, 20))

                              TRUE, FALSE
```

Example 2. `x` is assumed to be divisible by `3` with remainder `1`:

```
>> assume(x, Type::Residue(1, 3))

                          3 Type::Integer + 1
```

Which properties has `x + 2` got?

```
>> getprop(x + 2)

                             3 Type::Integer
```

`x` is an integer, but it may be odd or not:

```
>> is(x, Type::Integer), is(x, Type::Odd)

                              TRUE, UNKNOWN
```

The optional subset of the integers restricts the possible values of `x`:

```
>> assume(x, Type::Residue(2, 4, Type::PosInt)):
   is(x > 0),
   is(x^2 >= 4)

                              TRUE, TRUE
```

Changes:

- `Type::Residue` is a new function.

`Type::SequenceOf` – type for testing sequences

With `Type::SequenceOf`, sequences with specified objects can be identified.

Call(s):

- `testtype(obj, Type::SequenceOf(obj_type <, min_nr <, max_nr>>))`

Parameters:

`obj`	—	any MuPAD object
`obj_type`	—	the type of the objects; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`
`min_nr`	—	the minimal number of objects as nonnegative integer
`max`	—	the maximal number of objects as nonnegative integer

Return Value: see `testtype`

Related Functions: `_exprseq`, `testtype`, `Type::ListOf`

Details:

- The call `testtype(obj, Type::SequenceOf(obj_type))` checks, whether `obj` is a sequence with elements of the given type `obj_type`. In that case, it `TRUE`, otherwise `FALSE`.
- A sequence has the domain type `DOM_EXPR` and the type `"_exprseq"`.
- This type expects one or more arguments `obj_type <, min_nr <, max_nr>>`.
- The two optional parameters `min_nr` and `max_nr` determine the minimum and maximum number of arguments of the analysed sequence, respectively. If the numbers are not be given, the number of elements of the sequence will not be checked. If only the minimum is given, the sequence must have at least `min_nr` elements for the test to succeed.
- This type does not represent a property.

Example 1. Is the given sequence a sequence of identifiers?

```
>> testtype((a, b, c, d, e, f), Type::SequenceOf(DOM_IDENT))
```

```
TRUE
```

Is the given sequence a sequence of at least five real numbers?

```
>> testtype((0, 0.5, 1, 1.5, 2, 2.5, 3), Type::SequenceOf(Type::Real, 5))
```

```
TRUE
```

Changes:

- No changes.

`Type::Series` – type for testing series

With `Type::Series`, series can be identified.

Call(s):

- `testtype(obj, Type::Series(s_type <, pt>))`

Parameters:

`obj`	—	any MuPAD object
`s_type`	—	the type of the series; one of `Puiseux`, `Laurent` and `Taylor`
`pt`	—	additional parameter to specify the series (only for `Taylor`)

Return Value: see `testtype`

Related Functions: `testtype`, `series`

Details:

- The call `testtype(obj, Type::Series(s_types))` checks, whether `obj` is a series of type `s_type, ...` and returns `TRUE`, if it holds, otherwise `FALSE`.
- The series must be computed by `series`, otherwise `Type::Series` cannot identify a series correctly.

- This type expects one or two arguments `s_type <, pt>`.

  `s_type` can be one of the types `Puiseux`, `Laurent` and `Taylor`. For `Taylor` series an optional second argument can be given to specify the point x_0 with the equation `x = x_0`.

Example 1. The following call returns a Puiseux series:

```
>> s := series(sin(sqrt(x)), x);
   type(s);
```

```
                          3/2    5/2
                   1/2   x      x         3
                  x    - ---- + ---- + O(x )
                         6      120
```

```
                        Series::Puiseux
```

```
>> testtype(s, Type::Series(Puiseux)),
   testtype(s, Type::Series(Laurent)),
   testtype(s, Type::Series(Taylor))
```

```
                      TRUE, FALSE, FALSE
```

Next, examine a Laurent series:

```
>> s := series(1/sin(x), x);
   type(s);
```

```
                                3
                     1   x   7 x       4
                     - + - + ---- + O(x )
                     x   6   360
```

```
                        Series::Puiseux
```

Note that, although, the type of `s` is again `Series::Puiseux`, this series is a Laurent series, which is a special case of Puiseux series:

```
>> testtype(s, Type::Series(Puiseux)),
   testtype(s, Type::Series(Laurent)),
   testtype(s, Type::Series(Taylor))
```

```
                      TRUE, TRUE, FALSE
```

Finally, a Taylor series is a Laurent series as well:

```
>> s := series(exp(1/z), z = infinity);
   type(s)
```

```
             1    1      1       1       1       / 1  \
          1 + - + ---- + ---- + ----- + ------ + O| -- |
              z      2      3       4        5    |  6 |
                  2 z    6 z    24 z    120 z     \ z  /
```

```
                        Series::Puiseux
```

```
>> testtype(s, Type::Series(Puiseux)),
   testtype(s, Type::Series(Laurent)),
   testtype(s, Type::Series(Taylor))
```

```
TRUE, TRUE, TRUE
```

Note that for Taylor series, you can also check the indeterminate used and the expansion point:

```
>> testtype(s, Type::Series(Taylor, z = infinity)),
   testtype(s, Type::Series(Taylor, x = infinity)),
   testtype(s, Type::Series(Taylor, z = 0))

                        TRUE, FALSE, FALSE
```

Changes:

- No changes.

`Type::SetOf` – type for testing sets

`Type::SetOf(obj_type)` describes sets of elements of type `obj_type`.

Call(s):

- `testtype(obj, Type::SetOf(obj_type <, min_nr <, max_nr>>))`

Parameters:

`obj`	—	any MuPAD object
`obj_type`	—	the type of the objects; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`
`min_nr`	—	the minimal number of objects as nonnegative integer
`max_nr`	—	the maximal number of objects as nonnegative integer

Return Value: see `testtype`

Related Functions: `testtype`, `Type::ListOf`, `Type::Union`, `DOM_SET`

Details:

- The call `testtype(obj, Type::SetOf(obj_type))` checks, whether `obj` is a set with elements of the given type `obj_type`. If it is, the function returns `TRUE`, otherwise `FALSE`.
- A set has the domain type `DOM_SET`.

- This type expects one or more arguments `obj_type <, min_nr <, max_nr>>`.

  The two optional parameters `min_nr` and `max_nr` determine the minimum and maximum number of elements in the analysed set. If the numbers are not be given, the number of elements in the set will not be checked. If only the minimum is given, the set must contain at least `min_nr` elements for the test to succeed.

- This type does not represent a property.

Example 1. Is the given set a set of identifiers?

```
>> testtype({a, b, c, d, e, f}, Type::SetOf(DOM_IDENT))

                                TRUE
```

Is the given set a set of at least five real numbers?

```
>> testtype({0, 0.5, 1, 1.5, 2, 2.5, 3}, Type::SetOf(Type::Real, 5))

                                TRUE
```

Example 2. `testtype` is used to select sets with exactly two idetifiers:

```
>> S := {{a}, {a, b}, {d, 1}, {0, d}, {e}, {d, e}}:
   select(S, testtype, Type::SetOf(DOM_IDENT, 2, 2))

                          {{d, e}, {a, b}}
```

Changes:

- No changes.

`Type::Singleton` – type to identify exactly one object

`testtype(x, Type::Singleton(y))` is equivalent to `bool(x = y)`.

Call(s):

- `testtype(obj, Type::Singleton(t_obj))`

Parameters:

`obj` — any MuPAD object
`t_obj` — any object to identify

Return Value: see `testtype`

Related Functions: `_equal`, `bool`, `testtype`, `Type::Union`

Details:

- The call `testtype(obj, Type::Singleton(t_obj))` is equivalent to `bool(x = y)`, but the latter is faster.
- `Type::Singleton` can be used to create combined types, especially in conjunction with `Type::Union`, `Type::Equation` and other types expecting type information for subexpressions (see example 2).
- This type does not represent a property.

Example 1. Check, if `x` is really `x`:

```
>> testtype(x, Type::Singleton(x))
```

```
                                TRUE
```

But the next call does the same:

```
>> bool(x = x)
```

```
                                TRUE
```

Example 2. `Type::Singleton` exists to create special testing expressions:

```
>> T := Type::Union(Type::Singleton(hold(All)), Type::Constant):
```

With the type `T` the option `All` and any constant can be identified with one call of testtype:

```
>> testtype(4, T), testtype(hold(All), T), testtype(x, T)
```

```
                          TRUE, TRUE, FALSE
```

But (e.g., in procedures) the following example works faster:

```
>> test := X -> testtype(X, Type::Constant) or bool(X = hold(All)):
   test(4), test(hold(All)), test(x)
```

```
                          TRUE, TRUE, FALSE
```

One way to test a list of options for syntactical correctness is the following:

```
>> T := Type::Union(
          // Name = "..."
          Type::Equation(Type::Singleton(hold(Name)), DOM_STRING),
          // Mode = n, n in {1, 2, 3}
          Type::Equation(Type::Singleton(hold(Mode)),
                         Type::Interval([1,3], Type::Integer)),
          // Quiet
          Type::Singleton(hold(Quiet))
        ):

>> testtype((Name = "abcde", Quiet), Type::SequenceOf(T))

                                TRUE
```

We only allow the values 1, 2, and 3 for `Mode`, however:

```
>> testtype((Quiet, Mode = 0), Type::SequenceOf(T))

                                FALSE
```

Obviously, it would be a good idea to tell the user which options we could not grok:

```
>> error("Unknown option(s): ".expr2text(
           select((Quiet, Mode = 0),
                  not testtype, Type::SequenceOf(T))))

 Error: Unknown option(s): Mode = 0

>> delete T, test:
```

Changes:

- No changes.

`Type::TableOfEntry` – type for testing tables with specified entries

`Type::TableOfEntry(obj_type)` describes tables with *entries* of type `obj_type`.

Call(s):

- `testtype(obj, Type::TableOfEntry(obj_type))`

Parameters:

`obj` — any MuPAD object
`obj_type` — the type of the entries; can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

Return Value: see `testtype`

Related Functions: `testtype`, `table`, `Type::TableOfIndex`

Details:

- The call `testtype(obj, Type::TableOfEntry(obj_type))` checks, whether `obj` is a table and all entries of this table are of the type `obj_type`. If both conditions are met, the call returns `TRUE`, otherwise `FALSE`.
- This type expects one argument `obj_type`.
- The entries of a table are the right hand sides of the operands of a table.
- This type does not represent a property.

Example 1. The following table uses identifiers as keys and integers as entries:

```
>> T := table(a = 1, b = 2, c = 3, d = 4):
   testtype(T, Type::TableOfEntry(DOM_INT))
```

```
                                TRUE
```

`Type::TableOfEntry` only checks the type of the entries, not the keys:

```
>> T := table(a = 1, b = 2, c = 3, d = 4):
   testtype(T, Type::TableOfEntry(DOM_IDENT))
```

```
                                FALSE
```

```
>> delete T:
```

Changes:

- No changes.

`Type::TableOfIndex` – type for testing tables with specified indices

`Type::TableOfIndex(obj_type)` represents tables with *indices* (keys) of type `obj_type`.

Call(s):

- `testtype(obj, Type::TableOfIndex(obj_type))`

Parameters:

`obj`	—	any MuPAD object
`obj_type`	—	the type of the indices; can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

Return Value: see `testtype`

Related Functions: `testtype`, `table`, `Type::TableOfEntry`

Details:

- The call `testtype(obj, Type::TableOfIndex(obj_type))` checks, whether `obj` is a table and all indices (keys) are of the type `obj_type`. If both conditions are met, the call returns `TRUE`, otherwise `FALSE`.
- This type expects one argument `obj_type`.
- The indices of a table are the left hand sides of the operands of a table.
- This type does not represent a property.

Example 1. The following table uses identifiers as keys and integers as values:

```
>> T := table(a = 1, b = 2, c = 3, d = 4):
   testtype(T, Type::TableOfIndex(DOM_IDENT))
```

```
                                TRUE
```

`Type::TableOfIndex` only checks the types of the keys of the table, so the following call returns `FALSE`:

```
>> T := table(a = 1, b = 2, c = 3, d = 4):
   testtype(T, Type::TableOfIndex(DOM_INT))
```

```
                                FALSE
```

```
>> delete T:
```

Changes:

- No changes.

`Type::Union` – type for testing several types with one call

`Type::Union(type1, type2, ...)` represents all objects having at least one of the types `type1, type2, ...`

Call(s):

- `testtype(obj, Type::Union(obj_types, ...))`

Parameters:

`obj`	—	any MuPAD object
`obj_types`	—	a sequence of types; a type can be an object of the library `Type` or one of the possible return values of `domtype` and `type`

Return Value: see `testtype`

Related Functions: `testtype`

Details:

- The call `testtype(obj, Type::Union(obj_types, ...))` checks, whether `obj` has the type of at least one of the given types `obj_types, ...` If such a type is found, the call returns `TRUE`, otherwise `FALSE`.
- The call `testtype(obj, Type::Union(obj_types, ...))` is thus equivalent to the call `_lazy_or(map(obj_types, x -> testtype(obj, x)))`, testing `obj` against all types in turn until one is found which matches.
- `obj_types, ...` must be a (nonempty) sequence of types (see `testtype`).
- This type does not represent a property.

Example 1. Check, whether the given object is a positive or negative integer:

```
>> testtype(2, Type::Union(Type::PosInt, Type::NegInt))
```

```
TRUE
```

`x` however, is neither a positive nor a negative number:

```
>> testtype(x, Type::Union(Type::Positive, Type::Negative))
                                FALSE
```

Example 2. `testtype` is used to select positive and negative integers:

```
>> SET:= {-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2}:
   select(SET, testtype, Type::Union(Type::PosInt, Type::NegInt))
                            {-2, -1, 1, 2}
>> delete SET:
```

Changes:

- No changes.

`Type::Unknown` – type for testing variables

`Type::Unknown` represents identifiers and indexed identifiers.

Call(s):

- `testtype(obj, Type::Unknown)`

Parameters:

`obj` — any MuPAD object

Return Value: see `testtype`

Related Functions: `testtype`, `indets`

Details:

- The call `testtype(obj, Type::Unknown)` checks, whether `obj` is an identifier or an indexed identifier with an integer index. If it is, the call returns `TRUE`, otherwise `FALSE`.
- An identifier has the domain type `DOM_IDENT`. An indexed identifier is an expression with type `_index` and two operands, the first of which is an identifier and the second one is an integer. A local variable is not of type `Type::Unknown`.
- This type does not represent a property.

Example 1. `Type::Unknown` accepts identifiers:

```
>> testtype(x, Type::Unknown)
```

```
TRUE
```

`x[0]` is an indexed identifier accepted by `Type::Unknown`:

```
>> testtype(x[0], Type::Unknown)
```

```
TRUE
```

The index must be an integer:

```
>> testtype(x[-1], Type::Unknown),
   testtype(x[1.0], Type::Unknown)
```

```
TRUE, FALSE
```

Changes:

- No changes.

`Type::Zero` – a type and a property representing zero

`testtype(obj, Type::Zero)` is equivalent to `iszero(obj)`. `Type::Zero` is a property, too, which can be used in an `assume` call.

Call(s):

- `testtype(obj, Type::Zero)`
- `assume(x, Type::Zero)`
- `is(ex, Type::Zero)`

Parameters:

`obj` — any MuPAD object
`x` — an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`ex` — an arithmetical expression

Return Value: see `testtype`, `assume` and `is`

Related Functions: `testtype`, `is`, `assume`, `Type::NonZero`

Details:

- The call `testtype(obj, Type::Zero)` is equivalent to `iszero(obj)`, which performs a syntactical test if `obj` is zero. If it is, the call returns `TRUE`, otherwise, `FALSE` is returned.
- The call `assume(x, Type::Zero)` marks the identifier `x` as zero.

  The call `is(ex, Type::Zero)` derives, whether the expression `ex` is zero (or this property can be derived).
- This type represents a property that can be used in `assume` and `is`.
- The call `assume(x = 0)` has the same meaning as `assume(x, Type::Zero)`.

Example 1. `testtype` determines the syntactical equality to zero:

```
>> testtype(0.0, Type::Zero)

                                TRUE

>> testtype(x, Type::Zero)

                                FALSE
```

Example 2. `Type::Zero` can be used within `assume` and `is`:

```
>> is(x, Type::Zero)

                              UNKNOWN
```

Assumption that `x` is zero:

```
>> assume(x, Type::Zero):
   is(x^2, Type::Zero)

                                TRUE
```

The next example shows the difference between `testtype` and `is`:

```
>> is(x, Type::Zero), testtype(x, Type::Zero)

                            TRUE, FALSE
```

Now the property of `x` is removed:

```
>> delete x:
   is(x, Type::Zero), testtype(x, Type::Zero)
```

```
                              UNKNOWN, FALSE
```

A positive number cannot be zero:

```
>> assume(x > 0):
   is(x, Type::Zero), testtype(x, Type::Zero)

                               FALSE, FALSE
```

But in the next example `x` could be zero:

```
>> assume(x >= 0):
   is(x, Type::Zero), testtype(x, Type::Zero)

                              UNKNOWN, FALSE

>> delete x:
```

Changes:

- No changes.