`property` — library for properties

Table of contents

Properties of identifiers

MuPAD offers the possibility to compute symbolically with identifiers.

```
>> sqrt(x^2)

                                2 1/2
                              (x )
```

```
>> a^c*b^c

                                c   c
                              a   b
```

Such expressions can only be simplified if restrictions about the possible values of variables involved are known.

To denote such restrictions, MuPAD offers *properties* that can be set for an identifier with the function `assume`. Most system functions take properties into account while computing and simplifying. Because this mechanism is relatively new, not all functions take care of properties as much as they could mathematically.

Some examples of properties are `Type::Real`, `Type::Positive`, and `Type::Integer`. Identifiers with these properties will be treated as unknown *real*, *positive* or *integer* numbers.

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

                             Type::Real
```

```
>> Re(x), Im(x), simplify(sqrt(x^2))

                          x, 0, x sign(x)
```

The assumption `assume(x > 0)` has the same meaning as `assume(x, Type::Positive)`. It implies that `x` is a real number.

```
>> assume(x > 0):
   sqrt(x^2)

                                  x
```

```
>> assume(x < 0):
   sqrt(x^2)

                                 -x
```

As the last example shows, with a new assumption the current properties of an identifier is overwritten. With an optional operation, the current properties can be kept and combined with other assumptions.

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

                            Type::Integer
```

```
>> assume(x > 0, _and)

                            Type::PosInt
```

Global property

It is possible to define a property of *all identifiers*. This property is called *global property*.

The global property can be defined by the function `assume` and deleted only by the function `unassume` (not with `delete`).

The protected identifier `Global` carries the global property.

All identifiers should be real:

```
>> assume(Type::Real)

                              Type::Real
```

Now all identifiers are real:

```
>> is(x, Type::Real), is(y, Type::Real), is(k + l + m, Type::Real)

                          TRUE, TRUE, TRUE

>> is(x^2 >= 0), Re(x), Im(x)

                              TRUE, x, 0
```

Individual properties of identifiers can be assumed:

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

                            Type::Positive
```

The individual properties are combined with the global property with `_and`. To show this fact, a new global property will be defined:

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

                             Type::PosInt

>> assume(x, Type::Negative):
   getprop(x)

                             Type::NegInt
```

The global property and the individual property can exclude one another, the result is the "empty property":

```
>> assume(Type::Positive):
   getprop(x)

                            property::Null
```

The library `property` contains additionally the four functions `assume`, `getprop`, `is` and `unassume`. These functions are exported by the system and can be used by their short name without the package name `property`. They are described in the standard library.

All Properties

There are four types of mathematical properties available in MuPAD:

- Basic number domains, such as the integers, the rational numbers, the real numbers, the positive real numbers, or the prime numbers,
- intervals in basic number domains,
- residue classes of integers, and
- relations between an identifier and an arbitrary expression.

The following table contains all predefined classes of properties:

Basic Number Domains	
`Type::Complex`	$\mathbb{C}$
`Type::Even`	$2\mathbb{Z}$
`Type::Imaginary`	$\mathbb{R}i$
`Type::Integer`	$\mathbb{Z}$
`Type::Negative`	$\mathbb{R}_{<0}$
`Type::NegInt`	$\mathbb{Z}_{<0}$
`Type::NegRat`	$\mathbb{Q}_{<0}$
`Type::NonNegative`	$\mathbb{R}_{\geq 0}$
`Type::NonNegInt`	$\mathbb{N}$
`Type::NonNegRat`	$\mathbb{Q}_{\geq 0}$
`Type::NonZero`	$\mathbb{C} \setminus \{0\}$
`Type::Odd`	$2\mathbb{Z}+1$
`Type::PosInt`	$\mathbb{N}_{>0}$
`Type::Positive`	$\mathbb{R}_{>0}$
`Type::PosRat`	$\mathbb{Q}_{>0}$
`Type::Prime`	prime numbers
`Type::Rational`	$\mathbb{Q}$
`Type::Real`	$\mathbb{R}$
`Type::Zero`	$\{0\}$
Intervals	
`Type::Interval(a, b, T)`	$\{x \in T \colon a < x < b\}$
`Type::Interval([a], b, T)`	$\{x \in T \colon a \leq x < b\}$
`Type::Interval(a, [b], T)`	$\{x \in T \colon a < x \leq b\}$
`Type::Interval([a], [b], T)`	$\{x \in T \colon a \leq x \leq b\}$
	`a,b`: expressions
	`T`: basic number domain
Residue Classes	
`Type::Residue(a, b)` or	$b\mathbb{Z}+a$
`b*Type::Integer + a`	`a,b`: integers
Relations	
`= b`	$\{b\}$
`<> b`	$\mathbb{C} \setminus \{b\}$
`< b`	$\mathbb{R}_{<b}$
`<= b`	$\mathbb{R}_{\leq b}$
`> b`	$\mathbb{R}_{>b}$
`>= b`	$\mathbb{R}_{\geq b}$
	`b`: expression

If `T` is a type specifier for a basic number domain, an interval, or a residue class from the middle column of this table, then `assume(x, T)` attaches the mathematical property "is an element of S" to the identifier `x`, where S is the corresponding set in the right column. Similarly, `is(ex, T)` checks whether the expression `ex` belongs to the set S mathematically. The syntax for relations is more intuitive: for example, `assume(x < b)` attaches the mathematical property “is less than b” to the identifier `x`, and `is(a < b)` checks whether the relation $a < b$ holds true mathematically for the expressions `a` and `b`.

There are often several equivalent ways to specify a property: for example, `>= 0`, `Type::NonNegative`, and `Type::Interval([0], infinity)` are equivalent properties. Similarly, `Type::Odd` is equivalent to `Type::Residue(1, 2)`. There are also members of the `Type` library that do not correspond to mathematical properties, e.g., `Type::ListOf`.

```
>> assume(x, Type::Interval([-1], [1])):
   getprop(2*x); getprop(10*x^2)

                    ]0, 2] of Type::Positive

                    ]0, 10] of Type::Positive

>> assume(x > 0): assume(x <= 10, _and):
   getprop(x)

                      ]0, 10] of Type::Real
```

`property::hasprop` – **does an object have properties?**

`property::hasprop(object)` tests, whether an object has properties.

Call(s):

- `property::hasprop(object)`
- `property::hasprop()`

Parameters:

`object` — any MuPAD object

Return Value: `TRUE` or `FALSE`

Related Functions: `assume`, `getprop`, `is`, `indets`, `unassume`

Details:

- `property::hasprop(object)` tests, whether the object has properties and returns `TRUE` if the object or any subexpression has a property, otherwise `FALSE`.
- `property::hasprop` always returns `TRUE` if the global property is defined.
- Compared with `getprop`, `property::hasprop` is a fast function and can be used to determine, whether an object has properties without using the slower functions `getprop` or `is`.
- In some cases, the function `is` can derive some aspects without any defined property (see example 3)!

Example 1. Does the expression `2*(x+1)` have any properties?

```
>> property::hasprop(2*(x + 1))

                              FALSE

>> assume(x > 0):
   property::hasprop(2*(x + 1))

                              TRUE

>> getprop(2*(x + 1))

                               > 2

>> delete x:
```

Example 2. Is the global property defined?

```
>> property::hasprop()
```

```
                            FALSE
```

If the global property is defined, `property::hasprop` returns always `TRUE`:

```
>> assume(Type::Real):
   property::hasprop(2*(x + 1)), property::hasprop()
```

```
                         TRUE, TRUE
```

```
>> property::hasprop(sin(2*x^sqrt(2)) + cos(2*x)^sqrt(2))
```

```
                            TRUE
```

```
>> unassume():
```

Example 3. `property::hasprop` returns `FALSE`, but `is` can determine an answer unequal to `UNKNOWN`:

```
>> property::hasprop(a + 1 > a)
```

```
                            FALSE
```

```
>> is(a + 1 > a)
```

```
                            TRUE
```

Changes:

- `property::hasprop` is a new function.

`property::implies` – test whether one property implies another

`property::implies(prop_1, prop_2)` tries to decide whether `prop_1` implies `prop_2` or its converse.

Call(s):

- `property::implies(prop_1, prop_2)`

Parameters:

`prop_1, prop_2` — any properties

Return Value: `TRUE`, `FALSE`, or `UNKNOWN`

Related Functions: `is`, `assume`, `property::simpex`, `bool`

Details:

- Both arguments must be properties; see `property` for an overview of all properties.
- The return value is `TRUE` if `prop_2` can be inferred from `prop_1`, and `FALSE` if `not prop_2` can be inferred from `prop_1`. The result is `UNKNOWN` if neither implication could be proved; this may be because neither holds, or because the property mechanism is too weak to find a proof.

Example 1. Every positive integer is positive:

```
>> property::implies(Type::PosInt, Type::Positive)

                                    TRUE
```

Some positive numbers are positive integers, and some are not:

```
>> property::implies(Type::Positive, Type::PosInt)

                                  UNKNOWN
```

No positive number is a negative integer:

```
>> property::implies(Type::Positive, Type::NegInt)

                                    FALSE
```

Changes:

- `property::implies` is a new function.

`property::Null` – the empty property

If the combination of two properties is not possible, the “empty property” `property::Null` is returned as result.

Call(s):

- `property::Null`

Related Functions: `assume`, `getprop`, `is`

Details:

- The logical combination of two properties results in the "empty" or "impossible" property if the properties contradict each other.

Example 1. The property positive contradicts the property negative:

```
>> Type::Positive and Type::Negative

                                        property::Null
```

The intersection of two intervals can be empty. The corresponding properties are derived to the empty property:

```
>> Type::Interval(-1, 0) and Type::Interval(0, 1)

                                        property::Null
```

Changes:

- `property::Null` is a new function.

`property::simpex` – simplify Boolean expressions

`property::simpex(ex)` simplifies the Boolean expression `ex`.

Call(s):

- `property::simpex(ex)`

Parameters:

`ex` — Relation of type `"_less"`, `"_ leequal"`, `"_unequal"`, `"_equal"`, or `"_in"`; or Boolean expression whose atoms are Boolean constants or relations

Return Value: an expression that is equivalent to the given expression

Related Functions: `is`, `assume`, `piecewise`

Details:

- The properties of the identifiers are used to replace the atoms of the Boolean expression `ex` by `TRUE` or `FALSE` if possible.
- `property::simpex` tries to collect relations involving the same identifier into a single Boolean atom.

Example 1. A relation that is true:

```
>> property::simpex(1 > 0)

                              TRUE
```

An expression that is false:

```
>> property::simpex(x > 1 and x < -1)

                              FALSE
```

Find out the strongest border:

```
>> property::simpex(x > 3 and x > PI and x > ln(23))

                             PI < x
```

Example 2. Simplification of relations with the same variable:

```
>> property::simpex(x >= 2 and x <= 2)

                              x = 2
```

`expand` expands nested expressions (`property::simpex` does not call `expand`):

```
>> property::simpex(expand(0 < x and x < 2 and
                           (-1 < x and x <= 0 or 3 < x and x <= 4)))

                              FALSE
```

Expressions that contains identifiers with properties can often be simplified:

```
>> assume(x > 0):
   property::simpex(x <> 0), property::simpex(x > 1)

                           TRUE, 1 < x
```

Also relations between identifiers are considered:

```
>> assume(x < y, _and):
   property::simpex(x < y and (x > y or x > 0) and y <> 0)

                              TRUE
```

Example 3. Expressions of type `_in` can be simplified:

```
>> delete x:
   property::simpex(x in R_ and x >= 0)

                         x in [0, infinity[

>> property::simpex(not x in R_ and not x in Z_)

                              not x in R_
```

Background:

- `property::simpex` uses the property mechanism to simplify expressions.

Changes:

- `property::simpex` is a new function.