reFORM Documentation

Release 0.1-alpha

Ben Ruijl

Mar 11, 2019

Contents:

This is the user documentation for the symbolic manipulation toolkit reFORM. The developer manual can be generated with `cargo doc --document-private-items`.

CHAPTER 1

Introduction

reFORM is a symbolic manipulation toolkit with the goal to handle expressions with billions of terms, taking up terabytes of diskspace. In the future, it may be an alternative to FORM.

At the moment reFORM is in early development and cannot handle a large workload yet. However, there are many components that work. Specifically, the C and Python API for multivariate polynomials is fully functional. For an overview of all the reFORM features, see the manual.

The basic element of reFORM is a term. Each term in the input is treated sequentially. The reason for this is that the expression (a collection of terms) may not fit in memory.

reFORM works with modules, which are contained in an `apply` statement. In every module, each term in the input will go through all the statements one by one until they are merged at the end. Choosing when to split a module is up to the user, and can greatly affect performance: when there are a lot of duplicate terms, double work can be avoided by splitting the operations over multiple modules. On the other hand, if there are millions of terms, the overhead of sorting may be larger than the gain.

CHAPTER 2

Installation

reFORM is written in Rust, so one first needs to install the Rust compiler. The easiest way is to install Rustup, using:

```
curl https://sh.rustup.rs -sSf | sh
```

For more information, see the official Rust installation guide.

The latest stable version can then be installed using:

```
cargo install reform
```

The reFORM source code can be obtained from Bitbucket:

```
git clone git@bitbucket.org:benruyl/reform.git
```

To compile in release mode, use:

```
cargo build --release
```

You will find the binary `reform` in `target/release`.

To compile reFORM with API support, please see the *API* section.

CHAPTER 3

Quick start

We will start with a simple reFORM program that adds one to an argument of a function:

```
expr F = f(5);

apply {
    id f(x?) = f(x? + 1);
    print;
}
```

This code creates an expression *F*, and applies a list of instructions (a *module*) to every term in the expression. This example will yield:

```
f(6)
```

Save the code in a file called `add.rfm` and use

```
reform add.rfm
```

to run it and check the result for yourself.

A big difference between reFORM and languages such as Mathematica and Maple is that every operation inside a module will be applied to each term independently.

If you want to run with multiple cores, you can specify them with the `-w` flag.

CHAPTER 4

Pattern matching

Pattern matching is an essential part of a term manipulation system. It provides a way to modify a term by its shape and relations instead of by the actual contents. We introduce *wildcards*, which are denoted as letters followed by a question mark, to match to variables, numbers, and subexpressions.

Note: To see the output for the following examples, either a `print` has to be added to the source code, or reFORM must be run with the `-v` command line option.

For example:

```
expr F = f(5);
apply {
    id f(x?) = f(x? + 1);
}
```

will match the wildcard `x?` to `5`, consequently add 1 to it, and yield

```
f(6)
```

Note: Contrary to Form, the question mark is repeated on the right hand side!

A wildcard will match any function argument if it occurs by itself

```
expr F = f(1+x,3);
apply {
    id f(x?,3) = x?;
}
```

yields

```
1+x
```

A pattern at the ground level (not inside functions) can match to a subpart of the factors:

```
expr F = f(1)*f(f(4*x));
apply {
    id f(f(y?)) = y?;
}
```

yields

```
f(1)*x*4
```

If the pattern contains a term with multiple wildcards, the number needs to match exactly.

```
expr F = f(x1*x2);
apply {
    id f(y1?*y2?) = f(y1,y2);
}
```

yields

```
1+x
```

So,

```
expr F = f(x1*x2*x3);
apply {
    id f(y1?*y2?) = f(y1,y2);
}
```

does not match. In this previous case, there are multiple options. `y1` could have matched to `x1` and to `x2`. The match that reFORM picks is deterministic. If you want to obtain *all* options, see the `id all` option.

A wildcard can be restricted to a certain set of options:

```
expr F = f(f(4))*f(f(3));
apply {
    id f(x1?{f(4)}) = f(x1);
}
```

will only match to `f(4)`. The restriction can be any expression. However, at the moment they are not allowed to include any wildcards. Additionally, for numbers you can use number ranges in the sets: `<=5,>=5,<5,>5` to match a number in a range relative to a reference number (5 in this example.)

```
expr F = f(1)*f(4);
apply {
    id f(x?{>3}) = f(x1 + 1);
}
```

will only change `f(4)`.

Fractional numbers are allowed, i.e., `f(x?{>1/2})` will work as intended.

A function name can also be a wildcard:

```
expr F = g(4);
apply {
    id f?(x?) = f?(x? + 1);
}
```

yields `g(5)`.

4.1 Ranged wildcards

The pattern matcher can also match ranges of function arguments using ranged wildcards. These wildcard have a question mark on the front: e.g., `?a`.

For example:

```
expr F = f(1,2,3,4);
apply {
    id f(?a,4) = f(?a);
}
```

yields

```
f(1,2,3)
```

Using a combination of ranged wildcards and wildcards, some complex patterns can be matched:

```
expr F = f(1,2,f(3),4)*f(1,2,f(3));
apply {
    id f(?a,x?,?b)*f(?c,x?,?d) = f(?a,?b,?c,?d);
}
```

yields

```
f(1,2,4,1,2)
```

Note that ranged wildcards can be empty.

4.2 Many-mode

The `many` option can be used to let reFORM apply a pattern to the input as often as possible.

```
expr F = x^2;
apply {
    id many x = 5;
}
```

yields

```
25
```

A more complicated example is shown below:

```
expr F = x*y^4*z;
apply {
    id many x?*y^2 = f(x?);
}
```

yields

```
f(x)*f(z)
```

4.3 Obtaining all matches

All matches can be obtained using the `all` option to `id`. For example:

```
expr F = f(1,2,f(x1*x2,x3*x4,x5*x6),x1*x3,x3*x5);
apply {
    id all f(1,2,f(?a,x1?*x2?,?b),?c,x1?*x3?) = f(x1?,x2?,x3?);
}
```

yields

```
f(x3,x4,x5)+f(x5,x6,x3)
```

CHAPTER 5

Variables

reFORM has variables (not to be confused with the algebraic variables of the input), which are expressions that must fit in memory. They are shared between terms and between the global and the local scope.

They start with a `$` sign.

For example:

```
$a = 5;
print $a;
```

yields `5`.

Variables can be used for many things as this chapter will show. Below we give an example where the variable gives each term a unique id (if not run in parallel):

```
expr F = x + y + z;

$counter = 0;
apply {
    Multiply f($counter);
    $counter = $counter + 1;
    print;
}
```

yields

```
x*f(1) + y*f(2) + z*f(3)
```

5.1 Pattern matching

Parts of a pattern can be stored in a dollar variable using *`matchassign`*:

```
expr F = f(x,1,2,3);

$a = 0;
apply {
    matchassign f(y?,?b) {
        $a = 2*y?*f(?b);
    }
}
print $a;
```

yields

```
2*f(1,2,3)*x
```

5.2 Control flow

Variables are also used as loop parameters in *for*:

```
$a = 0;
for $i in 1..3 {
    $a = $a + $i;
}
print $a;
```

yields 2.

5.3 Indexing

Variables can be indexed as if they were functions. Combined with loops this is very flexible:

```
for $i in 1..3 {
    $a[$i+x,2] = $i;
}

$b = $a[2+x,4] + f(x);

inside $b {
    id f(x?) = $a[1+x?,2];
    id $a[x?,?a,y?] = $a[x?,?a,y?-2];
}

print $b;
```

yields 3.

As this example shows, variables will be automatically substituted.

5.4 Between modules

Variables can be used to collect global information about an expression, like the maximum value of a certain functions argument over all the terms. This information can be used in a later module. For example:

```
expr F = f(10) + f(20) + f(30);
$a = 0;

apply {
    matchassign f(x?) {
        $a = x?;
    }

    maximum $a;
}

apply {
    id f(x?) = f($a);
}
```

yields `3*f(30)`. This piece of code stores the maximum value of `$a` over all terms (see *`maximum`*). In the next module, `$a` will be set to 30.

5.5 Command line arguments

Dollar variables can be specified on the command line using the *-d* flag. Using the `$` is optional and will be added automatically.

For example:

```
reform -d $a=5,b=6,c=1+x,d="1 + t^2" test.frm
```

CHAPTER 6

API

Certain features of reFORM can be used inside other programs and programming languages using its API. We expose a *Python API* and a *C API*.

6.1 Python API

To compile the reFORM Python 3 library, compile with the `python_api` feature:

```
cargo build --release --features=python_api
```

This will produce a `libreform.so` (Linux), `libreform.dylib` (Mac OS), or `libreform.dll` (Windows) in `target/release`. Rename this file to `reform.so`, place it in the same folder as your Python script, and simply import it.

Note: On Mac OS, it could be that the code does not compile. A fix is to remove `features = ["extension-module"]` from `Cargo.toml`, which will mean that the Python 3 version that the library supports, is fixed.

An example Python program:

```
import reform

vi = reform.VarInfo()
a = reform.Expression("x+y^2", vi)
b = reform.Expression("z", vi)
c = a * b

print("c: ", c, ", c expanded: ", c.expand())

d = c.expand().id("x", "1+w", vi)
print("Substituted x->1+w: ", d)
```

An example Python program showing the polynomial API:

```
import reform

vi = reform.VarInfo();
a = reform.Polynomial("1+x*y+5", vi)
b = reform.Polynomial("x^2+2*x*y+y", vi)
g = a + b

ag = a * g
bg = b * g

rat = reform.RationalPolynomial(ag, bg)
print('ag/bg:', rat)
print('gcd:', ag.gcd(bg))
```

Polynomials can be converted to generic expressions and vice-versa with `to_expression()` and `to_polynomial()`. The latter function will fail if the expression is not a polynomial.

6.2 C API

To compile the reFORM C library, compile with the `c_api` feature:

```
cargo build --release --features=c_api
```

Then, compile your C code as follows:

```
gcc --std=c11 -o gcd gcd.c -L target/release/ -lreform
```

To run the C code, add the library to the path:

```
LD_LIBRARY_PATH=target/release/
```

An example C program:

```
#include <stdio.h>
#include <stdint.h>

typedef struct polynomial Polynomial;
typedef struct varinfo VarInfo;

extern VarInfo * polynomial_varinfo();
extern void polynomial_varinfo_free(VarInfo *);

extern Polynomial * polynomial_new(const char *expr, VarInfo*);
extern void polynomial_free(Polynomial *);
extern Polynomial * polynomial_clone(Polynomial *);
extern char * polynomial_to_string(Polynomial *);
extern void polynomial_string_free(char *);
extern Polynomial * polynomial_add(const Polynomial *, const Polynomial *);
extern Polynomial * polynomial_mul(const Polynomial *, const Polynomial *);
extern Polynomial * polynomial_sub(const Polynomial *, const Polynomial *);
extern Polynomial * polynomial_div(const Polynomial *, const Polynomial *);
extern Polynomial * polynomial_neg(const Polynomial *);
extern Polynomial * polynomial_gcd(const Polynomial *, const Polynomial *);
```

(continues on next page)

(continued from previous page)

```c
extern RationalPolynomial * rationalpolynomial_new(const Polynomial *, const␣
↪Polynomial *);
extern void rationalpolynomial_free(RationalPolynomial *);
extern Polynomial * rationalpolynomial_clone(Polynomial *);
extern char * rationalpolynomial_to_string(RationalPolynomial *);
extern Polynomial * rationalpolynomial_neg(const RationalPolynomial *);
extern RationalPolynomial * rationalpolynomial_add(const RationalPolynomial *, const␣
↪RationalPolynomial *);
extern RationalPolynomial * rationalpolynomial_mul(const RationalPolynomial *, const␣
↪RationalPolynomial *);
extern RationalPolynomial * rationalpolynomial_div(const RationalPolynomial *, const␣
↪RationalPolynomial *);
extern RationalPolynomial * rationalpolynomial_sub(const RationalPolynomial *, const␣
↪RationalPolynomial *);


int main(void) {
        VarInfo *vi = polynomial_varinfo();
        Polynomial *a = polynomial_new("1+x*y+5", vi);
        Polynomial *b = polynomial_new("x^2+2*x*y+y", vi);
        Polynomial *g = polynomial_add(a, b);

        Polynomial *ag = polynomial_mul(a, g);
        Polynomial *bg = polynomial_mul(b, g);

        Polynomial *gcd = polynomial_gcd(ag, bg);

        char *str = polynomial_to_string(gcd);
        printf("gcd: %s\n", str);

        RationalPolynomial *rat = rationalpolynomial_new(ag, bg); // g wil be removed
        char *s = rationalpolynomial_to_string(mrat);
        printf("ag/bg: %s\n", s);

        polynomial_string_free(s);
        polynomial_string_free(str);
        rationalpolynomial_free(rat);
        rationalpolynomial_free(mrat);
        polynomial_free(a);
        polynomial_free(b);
        polynomial_free(g);
        polynomial_free(ag);
        polynomial_free(bg);
        polynomial_free(gcd);
        polynomial_varinfo_free(vi);
}
```

CHAPTER 7

Reference Guide

7.1 Procedures

A procedure is a code block that will be inlined at the call-site.

proc name(args; localargs) { [statements] }

Parameters

- **args** – arguments to the function
- **localargs** – variables local to the procedure. They will shadow existing variables.

Define a code block that will be placed inline in the code when called with *call*. All variables in the arguments `args` are replaced. `localargs` will shadow arguments from the outer scope.

The code block can contain *apply* statements.

```
proc derivative(x, n) {
    for $i in 1..(n+1) {
        id x^m? = m? * x^(m? - 1);
    }
}

expr F = u^5;

apply {
    call derivative(u, 2);
}
```

yields

```
20*u^3
```

7.2 User-defined functions

Users can define their own functions in the global scope with the following statement:

`fn name(args) = expression;`

Parameters

- **`name`** – Name of the function
- **`args`** – Arguments to the function
- **`expression`** – The resulting expression

Replace the function `name` with `expression` where all occurences of the `args` are replaced by the given function arguments.

Note: Functions in already existing expressions will not be automatically substitued when they are defined as custom functions at a later stage. Use `id myfunc(?a) = myfunc(?a);` to trigger the substitution.

```
fn factorial(n) = ifelse_(n > 0, n * factorial(n-1), 1);
$a = factorial(10);
print $a;
```

yields

```
3628800
```

7.3 Statements

`apply [name for F1,...,F2 exclude F3,...,] { [statements] };`

Parameters

- **`name`** – Optional name of the module
- **`statements`** – A list of statements that will be applied

Apply a list of statements (a module) to all active expressions. If `for F1,...,F2` is specified, the module is only applied to these expressions. It is also possible to apply the module to all expressions excluding some by using `excluding F3,...`. The `apply` statement cannot be nested.

For example:

```
expr F = f(5);
apply {
    id f(x?) = f(x? + 1);
    id f(6) = f(3);
}
```

The statements will be processed term by term.

`argument f1,f2,... { [statements] }`

Parameters

- **`f1,...`** – Functions the statements should be applied to.
- **`statements`** – Statement block to be executed on function arguments

Execute a block of statements on the arguments of specific functions.

```
expr F = f(1+x,y*x);

apply {
    argument f {
        id y = 5;
    }
}
```

yields

```
F = f(1+x,5*x)
```

`assign x = expr;`

Parameters

- **`x`** – A variable
- **`expr`** – A reFORM expression

Assign the expression to the variable `x`.

```
$a = 1 + x;
print $a;
```

yields

```
1 + x
```

`attrib f = Linear + NonCommutative + Symmetric;`

Parameters

- **`f`** – A function name.

Assign attributes to a function. At the moment the options are `Linear`, `NonCommutative`, and `Symmetric`. Multiple options can be given with a `+`.

```
expr F = f(x, y);

attrib f = Linear;

apply {
    id f(x1?,x2?) = f(x1?+2,x2?+5);
}
```

yields

```
+f(x,y)
+f(x,5)
+f(2,y)
+f(2,5)
```

`call proc(args);`

Parameters

- **`proc`** – A procedure
- **`args`** – Arguments to the procedure

Call a procedure (see *Procedures*) with arguments.

```
procedure derivative(x, n) {
    for $i in 1..(n+1) {
        id x^m? = m? * x^(m? - 1);
    }
}

expr F = u^5;

apply {
    call derivative(u, 2);
}
```

yields

```
u^3*20
```

`collect fn;`

Parameters

- **`fn`** – A function name.

If this statement is called *inside* a module, it will wrap the entire term in a function `fn`. if this statement is called outside the module, it will wrap the entire expression in a function `fn`. The latter is only possible if the expression fits in memory.

Note: The collect statement must currently be placed before the apply block. This will be fixed in the future.

```
expr F = (1+x)^4;

collect f;
print;
apply {
    expand;
}
```

yields

```
+f(x*4+x^2*6+x^3*4+x^4+1)
```

`discard;`

Discard the current term.

```
expr F = x + y;
apply {
    if match(x) {
        Discard;
    }
}
```

yields

```
y
```

`expand;`

Expand all structures. For example, \` `(1+x)^5`\`, and \` `(1+x)*(1+y)`\` will be completely written out.

```
expr F = (1+x)^2*(1+y);
apply {
    expand;
}
```

yields

```
+x*y*2
+x*2
+x^2
+x^2*y
+y
+1
```

expr name = expression;

Parameters

- **name** – The name of a new expression
- **expression** – Any valid reFORM expression.

Create a new *expression*. An expression is processed term-by-term and can be larger than memory. Use *apply* to operate on the terms of the expression.

extract $i x1,...,xn;

Parameters

- **$i** – A reFORM variable.
- **x1,...,xn** – A list of algebraic variables.

Construct a Horner scheme in the variables `x1` to `xn` for the expression in variable `$i`.

```
$a = x + x*y + x*y*z + y*z + x^2 + x^2*y + 2;

extract $a x,y;
print $a;
```

yields

```
(y+1)*x^2+y*z+2+((z+1)*y+1)*x
```

fn name(args) = expression;

See *User-defined functions*.

for i in lb..ub { [statements] };
for i in {s1,s2,...} { [statements] };

Parameters

- **i** – The loop variable.
- **lb..ub** – A numerical range.
- **{s1,s2,...}** – A list of expressions.

Loop over a numerical range or over a list of expressions. Loops can be made both inside and outside of modules.

```
expr F = f(2);

for $i in 1..4 {
    print;
    apply {
        id f($i) = f($i+1);
    }
}
```

yields

```
F = f(2);
F = f(3);
F = f(4);
```

id lhs = rhs;

Parameters

- **lhs** – Any valid reFORM expression with *wildcards*.
- **rhs** – Any valid reFORM expression with *wildcards*.

Apply the lhs to an active term (therefore an `id` statement needs to be in an `inside` or `apply` block (module).

See *Pattern matching* for the patterns that are allowed to match.

For example:

```
expr F = f(5);

apply {
    id f(x?) = f(x? + 1);
}
```

if cond { [statements] } [else { [statements] }]
if match(expr) { [statements] } [else { [statements] }]
if defined(dollar) { [statements] } [else { [statements] }]

Parameters

- **cond** – A boolean condition
- **match(expr)** – A test to see if an expression matches
- **defined(dollar)** – A test to see if a dollar variable is defined
- **statements** – Statement block to be executed

Only execute if a condition holds. If there is an `else` block, that will only be executed if `cond` does not hold.

The condition can test if a pattern exists (see frm:st:*id*) using the `match` option. The condition can also be a comparison of two expressions, i.e., `<=, >=, <, >, ==`.

Note: Inequalities use reFORM's internal ordering which may give unexpected results.

```
expr F = f(1);

apply {
    if match(f(1)) {
```

(continues on next page)

(continued from previous page)

```
        id f(1) = f(2);
    } else {
        id f(x?) = f(1);
    }

    if defined($a) {
        Multiply $a;
    }

    if f(1) < f(2) {
        id f(2) = f(3);
    }
    print;
}
```

yields

```
f(3)
```

inside x1,x2,... { [statements] }

Parameters

- **x1, ...** – Variables the statements should be applied to.
- **statements** – Statement block to be executed on the terms in variables.

Execute a block of statements on specific variables.

```
$x = 1 + x + y*x;

inside $x {
    id x = 5;
}
print $x;
```

yields

```
6 + 5*y
```

matchassign pattern { [assigns] };

Parameters

- **pattern** – A pattern to match the current expression to.
- **assigns** – A list of *assign* statements.

Match the current term and use the matched wildcards in the assignment of dollar variables.

```
expr F = f(x,1,2,3);

$a = 0;
$b = 0;
apply {
    matchassign f(y?,?b) {
        $a = 2*y?*f(?b);
        $b = y?^5;
    }
```

(continues on next page)

(continued from previous page)

```
}
print $a,$b;
```

yields

```
2*f(1,2,3)*x
x^5
```

maximum x;

Parameters

- **x** – A variable

Get the maximum of the variable x over all terms in the module.

```
$a = 0;
apply {
    if match(f(1)) {
        $a = 2;
    } else {
        $a = 1;
    }

    maximum $a;
}
print $a;
```

yields

```
2
```

multiply expr;

Parameters

- **expr** – An expression to multiply.

Multiply the expression into the current active term. `Multiply` can only be used in a module.

```
expr F = y;
apply {
    Multiply 1 + x;
}
```

yields

```
y*(1+x)
```

print [format] [vars];

print [format] format_string;

Parameters

- **format** – Optional format for printing. It can either be `Form` or `Mathematica`.
- **vars** – A list of variables to print.
- **format_string** – a list of variables to print

Print objects or a formatted string to the screen.

If the `Print` statement without arguments `vars` or `format_string` is used in a module, the current term is printed. If it is used outside a module without these arguments, it will print all active expressions.

The `format` option can be used to format the terms in a way such that it is compatible with other software. The current supported options are `Form` (default) and `Mathematica`.

If a list of variables `vars` is specified, each variable will be printed on a new line. If a format string is specified, the formatted string is printed. Variables and special objects can be printed by putting them between `{ }` in the format string. Special objects are:

- `{data_}`: print the current date and time
- `{time_}`: print the current time
- `{term_}`: print the current term
- `{$a}`: print the value of `$a`

```
$a = f(x);
print mathematica $a;

expr F = 1 + x;
apply {
    print; // print the current term
    print "{date_}: current term={term_}, $a={$a}";
}
print; // print F
```

procedure name(args; localargs) { [statements] }

See *Procedures*.

repeat { [statements] }

Parameters

- **statements** – Statement block to be repeated until no terms change anymore.

Repeat a block of statements until the term does not change anymore.

The code below does a naive Fibonacci series evaluation. The repeat block will continue until none of the three *id* statements match.

```
expr F = f(30);

apply {
    repeat {
        id f(x?{>1}) = f(x? - 1) + f(x? - 2);
        id f(1) = 1;
        id f(0) = 0;
    }
}
```

yields

```
F = f(1,x,2*y)
```

replaceby expr;

Parameters

- **expr** – An expression

Replace the current term by `expr`.

```
expr F = x*y + y;
apply {
    if match(x) {
        ReplaceBy z;
    }
}
```

yields

```
y + z
```

splitarg fn;

Parameters

- **fn** – A function

Split a subexpression in a function argument into new function arguments. For example:

```
expr F = f(1+x+2*y);

apply {
    splitarg f;
}
```

yields

```
F = f(1,x,2*y)
```

symmetrize fn;

Parameters

- **fn** – A function name.

Symmetrize the function arguments based on reFORM's internal ordering.

```
expr F = f(3,2,x,1+y,g(5));

apply {
    symmetrize f;
}
```

yields

```
f(g(5),y+1,x,2,3)
```

7.4 Functions

delta_(x1)

Parameters

- **x1** – A reFORM expression

Returns 1 if `x1` is 0. If it is a number other than 0, it will return 0.

If `x1` is not a number, nothing happens.

```
expr F = delta_(0)*x + delta_(1)*y + delta_(x);
```

yields

```
x + delta_(x)
```

gcd_(p1, p2)

Parameters

- **p1** – A multivariate polynomial with integer numbers as coefficients
- **p2** – A multivariate polynomial with integer numbers as coefficients

Compute the greatest common divisor of two multivariate polynomials with integer numbers as a coefficient.

If the arguments are not valid polynomials, no replacement will be made.

```
expr F = gcd_(100+100*x-90*x^3-90*x^4+12*y+12*x*y+3*x^3*y^2+3*x^4*y^2,
              100-100*x-90*x^3+90*x^4+12*y-12*x*y+3*x^3*y^2-3*x^4*y^2);
```

yields

```
+x^3*y^2*3
+x^3*-90
+y*12
+100
```

ifelse_(cond, truebranch, falsebranch)

Parameters

- **cond** – A comparison, i.e., `$a < 2`
- **truebranch** – An expression that will be the result of the function if the condition is true
- **falsebranch** – An expression that will be the result of the function if the condition is false

Return `truebranch` if the condition `cond` is true and `falsebranch` if it is false. At the moment `cond` should be a comparison between expressions. If the expressions are both numbers, all both equality and inequality tests are evaluted. In all other cases, only an equality test will be evaluated.

Note: The expressions in both branches are not normalized (simplified), since that will take extra work (only one of the branches should be executed) and could cause infinite loops. As a result, pattern matching on the arguments of `ifelse_` will likely not work.

```
expr F = f(5);
apply {
    id f(n?) = ifelse_(n? <= 6, n? + 10, n?);
}
```

yields

```
15
```

list_(i, lb, ub, expr)

Parameters

- **i** – A variable used as a counter
- **lb** – A numeric lower bound for i
- **ub** – A numeric upper bound for i

Return a list of expr with i going from lb to (and including) ub. This function will only be replaced when it is a function argument.

```
expr F = f(1,2,list_($i,2,5,$i^2),3,4);
```

yields

```
f(1,2,4,9,16,25,3,4)
```

nargs_(a1,...,an)

Parameters

- **a1,...,an** – A list of expressions

Returns the number of arguments the function has. It is especially useful in combination with the *ranged wildcards*.

```
expr F = f(1,2,3,4,5);

apply {
    id f(?a) = nargs_(?a);
}
```

yields

```
5
```

prod_(i, lb, ub, expr)

Parameters

- **i** – A variable used as a counter
- **lb** – A numeric lower bound for i
- **ub** – A numeric upper bound for i

Return the product of expr with i going from lb to (and including) ub.

```
expr F = prod_($i, 2, 5, $i^2);
```

yields

```
14400
```

rat_(num, den)

Parameters

- **num** – A multivariate polynomial with integer numbers as coefficients
- **den** – A multivariate polynomial with integer numbers as coefficients

The rat_ function can be used to have a ratio of multivariate polynomials as a coefficient . It will compute multivariate gcds to make sure the fraction does not grow more than necessary.

If the arguments are not valid polynomials, no replacement will be made.

```
expr F = rat_(x^2+2*x+1,1)*rat_(1,1+x)+rat_(2,1);
```

yields

```
rat_(3+x,1)
```

sum_(i, lb, ub, expr)

Parameters

- **i** – A variable used as a counter
- **lb** – A numeric lower bound for `i`
- **ub** – A numeric upper bound for `i`

Return the sum of `expr` with `i` going from `lb` to (and including) `ub`.

```
expr F = sum_($i, 2, 5, $i^2);
```

yields

```
54
```

takearg_(k,a1,...,an)

Parameters

- **k** – The index of the argument to take
- **a1,...,an** – Arguments

Return the `k` th argument of the list `a1,...,an` . If the index is out of bounds, no substitution takes place.

```
expr F = takearg_(2, x1, x2, x3);
```

yields

```
x2
```

CHAPTER 8

Index

- genindex

Index