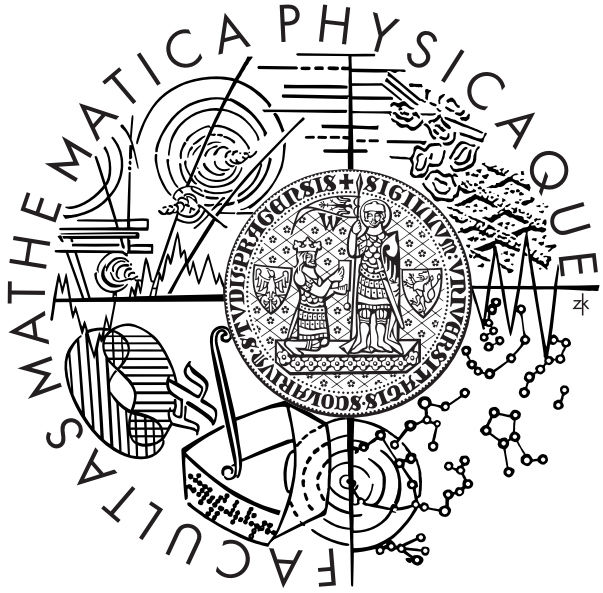
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Ján Majdan

C++ library for symbolic manipulation

Supervisor: RNDr. Tomáš Holan, Ph.D.
Department of Software and Computer Science Education

Specialization: General Computer Science

2009

I would like to thank my supervisor, RNDr. Tomáš Holan Ph.D., for his time and his flexible consultations. I also would like to thank my consultant, Mgr. Ondřej Čertík, for his good advice and ideas that strongly affected my work.

Furthermore, I would like to thank my friends Ing. Pavol Korbel, Bc. Vlasta Poliačková, Eva Fedurcová and Krisitína Hajniková for some comments and corrections.

I declare that I wrote the submitted thesis myself, using only referenced sources of information. I agree with lending and publishing the thesis.

Prague, August 5, 2009 Ján Majdan

Contents

Názov práce: C++ knižnica pre symbolickú manipuláciu
Autor: Ján Majdan
E-mail autora: jan.majdan@gmail.com
Katedra: Kabinet software a výuky informatiky
Vedúci bakalárskej práce: RNDr. Tomáš Holan, Ph.D.
E-mail vedúceho: Tomas.Holan@mff.cuni.cz
Konzultant: Mgr. Ondřej Čertík
E-mail konzultanta: ondrej@certik.cz

Abstrakt: Cieľom tejto bakalárskej práce je návrh a implementácia knižnice pre symbolickú manipuláciu s matematickými výrazmi. Ďalším zámerom práce je opísanie a odôvodnenie použitých metód a techník. Vyvinutá knižnica by mala byť jednoducho rozšíriteľná a modifikovateľná. Na druhej strane, táto knižnica má byť použiteľná pre dlhé a zložité výpočty. Demonštrácia riešenia oboch spomenutých požiadaviek, ktoré sú protichodné, je tiež súčasťou predloženej práce. V neposlednom rade sú v tomto texte poskytnuté návody a príklady použitia tejto knižnice.

Kľúčové slová: symbolická manipulácia, systém počítačovej algebry, C++ knižnica, zjednodušovanie matmatických výrazov

Title: C++ library for symbolic manipulation
Author: Ján Majdan
Author's e-mail address: jan.majdan@gmail.com
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Tomáš Holan, Ph.D.
Supervisor's e-mail address: Tomas.Holan@mff.cuni.cz
Consultant: Mgr. Ondřej Čertík
Consultant's e-mail address: ondrej@certik.cz

Abstract: The aim of the bachelor thesis is the design and implementation of a library for symbolic manipulation with mathematical expressions, and the other intention of the work is the description and substantiation of used techniques and methods. The developed library should be easily expandable and modifiable. On the other hand, the library should be usable for long and complex computation. Demonstration of solutions for both of the mentioned requirements, that often require opposite approaches, is also part of the submitted work. The last, but not least, there is the topic that is concentrated on some hints and examples of using the library.

Keywords: symbolic manipulation, computer algebra system, C++ library, simplification of mathematical expressions

Chapter 1

Introduction

The idea of using computers for mathematical computations arises when long and complicated calculations should be done. Using computers as tools for solving mathematical problems is not a new conception. In fact, this was the primary reason for constructing computers in the past. This intention is often forgotten in the times of computer games, chats, social networks etc.

There are many mathematical problems and also numerous methods for solving them by computers. Every particular part of Mathematics has its own software tools that are often depended on each other. Numerical mathematics is probably the most frequently used branch of mathematics in the computer world. Numerical methods provide a wide background for many other fields of mathematics. Methods that are based on iterations and approximations frequently have a polynomial complexity and the precision of results is adjustable.

However, there are two sides to each coin. Approximate results are not always acceptable. Also cooperation of the methods is necessary to solve difficult problems, but partial outcomes can become too long and complicated for following computations. These difficulties are solved by symbolic manipulations. Knowledge of mathematical expression structures and modification of the structures can sometimes simplify problems and it can also help in finding to a way to the exact results. Very good performance is achieved by software that uses the symbolic manipulations as the first automatic step of computation.

It is evident that correct, quick and clever symbolic manipulations are the fundamentals for powerful mathematical software. I have implemented basics of a symbolic manipulation library for my academic year project. This project is ensued and extended by this bachelor thesis. The library has been written in C++ and named sympy–cpp. The name was chosen by Sympy [1] (other symbolic manipulation package), because the easy and efficient scheme of this package has been an inspiration for the design of sympy–cpp.

The contents of the chapters are introduced in the following part:

2 Symbolic manipulation

This chapter provides an introduction to symbolic manipulation. The main goal is to describe basic capabilities and utilities of symbolic manipulation packages and specifically a subset of them that has been implemented in sympy–cpp. The second part of the section contains discussion about technical realization of the symbolic manipulation packages.

3 Usage of sympy–cpp

This chapter contains a quick and easy user's guide and some instructions about the compilation and installation of the library. The chapter also includes some examples of using sympy–cpp.

4 Inside sympy–cpp

This chapter concentrates on inner principles operating within the library. The chapter is not a summary of all classes, functions and structures, but it contains a detailed description of the important classes, their mutual relations, used algorithms and reasons for their usage.

5 What is next

This section contains a discussion about plans of future development of sympy–cpp. Also, elements that are missing in the library or should be done better are outlined.

6 Summary

The goal of the chapter is the presentation of sympy–cpp as a working symbolic library. The chapter contains performed demonstration of the library's performance and comparison with the other packages.

Chapter 2

Symbolic manipulation

Symbolic manipulations are modifications within structures of mathematical expressions. Replacing and reordering of expressions belong among the transformations that are used to change expressions or subexpressions from one form to another. Sequences of the transformations create symbolic manipulations.

2.1 Capabilities of symbolic manipulation packages

The symbolic manipulation packages keep hold of a set of symbolic manipulations. This set defines capabilities and performance of this package. The abilities can be distributed into three groups.

The first group is created from fundamental automatic actions of the symbolic system. These actions should be embedded in a core of every system. For example, there are automatic simplifications, arithmetic operations and operations with long or rational numbers.

The second group contains important functionality which does not work automatically. Substitutions, differentiation, integration, factorization, progressive simplification, etc are located here.

Finally, the last group will be the biggest division, as it contains facilities for anything. There are special mathematical tools for particular problems, some transformation tools for neat printing of math and so on. Many of these functions are not symbolic manipulations anymore.

Sympy–cpp is the mini library which covers the first group of fundamental functions and a part of the second group. Sympy–cpp has implemented automatic simplification of expressions, substitution, expansion, differentiation, Taylor series and arithmetic is solved by an external library. The description and disquisition of fundamental functions abilities and additional functions follows.

2.2 Automatic fundamental functionality

The automatic abilities are essential components of each symbolic package. They work after and during the process of computation. Their implementation is necessary and should be fast.

2.2.1 Automatic simplification

The automatic simplification is the transformation of mathematical expressions by any rules to simpler forms. The selection of utilized rules is very important. The rules have to ensure adequate simplification and they should be simple enough for easy and fast implementation.

1. Addition and subtraction of identical subexpressions or their multiples.
   $2 + x + Sin(x) + Sin(x) - 3 - 5x \approx -1 - 4x + 2Sin(x)$
2. Multiplication and division of identical subexpressions or their powers.
   $2x3(-x) + \frac{(y+3)z(3+y)^w}{z^3} \approx -6x^2 + \frac{(3+y)^{1+w}}{z^2}$
3. Evaluation function in opportune cases.
   $2 + Cos(Sin(0)) \approx 3$
4. Elimination of zero and expressions equal to zero.
   $3 - x - 3 + (x + z)(23 + y)Sin(0) \approx -x$
5. Simplification of powers.
   $x^{5-x-3+x} + (x^2 + 2)^{x/x} + 2 + (3x)^0 \approx 5 + 2x^2$
6. Addition of exponents of powers that have the same bases.
   $x^{4+a}(-y^3)x^b y^{-2} \approx x^{4+a+b}(-y)$
7. Multiplication of exponents in power of power.
   $(x^3)^c + ((y + 2)^2)^{b+1} \approx x^{3c} + (2 + y)^{2(1+b)}$

Enumeration of simplifying transformations is not complete. Expansion, factorization or any other modifications can achieve a better simplifying effect, but worse results are also possible. Automatic simplification is always applied and a possibility to switch off or hand manage this process does not exist. Such complex manipulations are not suitable for automatic simplification. All unapplied simplification methods should be available in the extension of the fundamental functions.

2.2.2 Arithmetic and numbers

Numbers and arithmetic are inherent ingredients of all mathematical software and such also symbolic manipulation software.

Execution of arithmetic is included in simplification and therefore it needs to be really fast. Representation of numbers by computers should be as close to the exact mathematical numbers as possible. Long numbers, real numbers, rational numbers and integers have to be designed for fast and precise computation. Also, they have to be compatible with each other.

Satisfaction of these requirements needs some sophisticated techniques. This dilemma of arithmetic is relevant for all symbolic packages. Some of the packages utilize benefits from the language of its implementation. On the other hand,

using these benefits often appears to be problematic, as these features are often too general and therefore they do not have satisfactory performance.

For small projects like sympy–cpp it is customary to use an external library for fast arithmetic.

2.3 Next functionality

Functions in this section are also important but not necessary for operation of the symbolic packages. They increase power and usability of the systems.

2.3.1 Substitution

Substitution replaces an expression by another expression. Combination of automatic simplification and substitution is a really strong instrument. The idea of substitution is general and thus there are more types of substitution.

1. A variable replaced by a number.
2. A variable replaced by a variable.
3. A variable replaced by an expression.
4. An expression replaced by a variable.
5. An expression replaced by an expression.

Each of these types of substitution has different capabilities and limitations. The first substitution is plain and safe. An increase in substitution abilities implies also an increase in hazardous usage. The construction of infinite recursion is enabled by simple substitution x by $-x$ into $1+x$ that creates infinite expressions series $1-x, 1+x, 1-x, \ldots$ where this substitution is repeatedly acceptable. Cases of such elementary recursions are easily detectable and fixable, but there are complex expressions and substitution which can invoke massive cyclic modification that are hardly detectable.

2.3.2 Expression expansion

This symbolic manipulation modifies multiplication of summation to summation of multiplication. The expression expansion can simplify or complicate expressions. For example:

1. $(x(y-\frac{1}{x})+y-y(x-\frac{1}{y})-x)(-x-y)(-1) \;\approx\; y^2-x^2$
2. $(x+y)^2(3+x^5)(y^x+x^y) \;\approx\; 3x^{2+y}+x^{7+y}+3y^{2+x}+6xy^{1+x}+3x^2y^x+x^7y^x+3x^yy^2+6x^{1+y}y+2x^{6+y}y+x^5y^{2+x}+2x^6y^{1+x}+x^{5+y}y^2$

It is evident, where the expansion is simplifying and where not from these examples. But it is not always so clear. Consequently, the expansion is available as a user's choice.

Expressions frequently contain powers, that have a sum in the base and an integer in the exponent, in which case it is possible to make the expansion by

binomial theorem [B.1] or multinomial theorem [B.2]. Such improvements of computation are strongly depended on implementation of arithmetic and numbers. Already an expression as simple as $(3 + a + b)^{100}$ needs arithmetic that operates with numbers that have more than a hundred digits.

2.3.3 Differentiation

Differentiation is a basic element of mathematical analysis and it is widespread tool used in many fields of mathematics and physics. The definition using limitation is not practical for symbolic systems. A better way is to use an established conversion table for basic mathematical functions and differentiations of composite functions are calculated by few simple rules.

1. $(f + g)' = f' + g'$
2. $(fg)' = f'g + fg'$
3. $(\frac{1}{f})' = \frac{-f'}{f^2}$
4. $(f \circ g)' = (f' \circ g)g'$

2.3.4 Taylor series

One of the most commonly used techniques is a approximation of a function as a power series that contains derivatives of this functions. Taylor series of the function $f(x)$ in the point a is equal to sum

$$\sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x - a)^k.$$

2.3.5 Integration

Integration is more difficult than differentiation, mainly regard to symbolic manipulation. There are more rules and integration is not as straightforward as differentiation.

2.3.6 Factorization

Factorization should be a reverse function to the expansion. But factorization often does not have a direct way to find an exact final form. One of the problems is to find roots of polynomials with grade greater than 3.

2.3.7 Progressive simplification

Progressive simplification is a combination of expansion, factorization and some tricks. Analysis of actions and quick deduction of complex manipulation plays a significant role in progressive simplification.

The list of basic functionality of the symbolic systems can be longer or the systems can implement only a part of the mentioned abilities. The primary requirement is that they should be clever, safe and fast implementations of selected features. Sympy–cpp does not implement progressive simplification, integration and factorization.

2.4 Technical realization of the symbolic manipulation systems

Numerous symbolic manipulation packages provide an impressive GUI and some programming tools. The offer of symbolic products is extensive. Commercial software (Maple, Mathematica, Matlab, Reduce, Mupad,...) and also free software (Maxima, Sympy, Sage, yacas, GiNaC, Giac,...) is available.

Users with little skills in programming can choose almost anything, but real programmers have particular demands that need to be satisfied.

Specially commercial products utilize their own programming language. This approach faces several problems. The first problem is the necessity of learning a new language, which is probably the smallest problem. The other defects are the absence of programming tools and the main restriction of this design is the fact that scripts/programs can run only on a computer where this particular symbolic software is installed. All this is too restrictive for standard programming.

Free symbolic manipulation packages are frequently created in languages[1] with strong background for symbolic manipulations and these languages are also provided as programming tools for users. However, small user base, relatively slow computation or a combination of both drawbacks do not make such packages the best alternative.

Of course, there are packages for programmers who need a fast and useful solution that is compatible and combinable with modern programming languages. Giac and GiNaC are representatives of C/C++ symbolic libraries and also Maplesoft provides C/C++ library in conjunction with program Maple. Unfortunately, the license of this library permits performing programs, that use this library, only on systems with Maple installed.

The background of C++ language is not ideal for implementation of symbolic manipulation package, but it is still feasible. The complexity of symbolic manipulation in C++ was shown in GiNac, which is impressive but unfortunately too complicated.

This project attempts to achieve an easy implementation and really smart and effective functionality of symbolic manipulation in C++. The second aim is to achieve sufficient speed of computation.

[1] most frequented are Prolog, Python and many dialects of Lisp

Chapter 3

Usage of sympy–cpp

The usage and installation of sympy–cpp is relatively simple. Some non-standard knowledge for usual users should be assumed. But the focus group of this project consists of programmers and people close to programming and computers. This focus group can easily obtain the required knowledge or they usually already have the information.

3.1 Installation

The installation consists of compilation of the source codes and the placement of the built library in a filesystem. Although the locations of libraries in diverse systems have usually standard positions, users can prefer different locations. Therefore the location of the created library and its header files is in the subfolder *sympy-cpp-lib* of the main folder with sympy–cpp source codes. After building of the library the users can copy the files wherever they need it.

3.1.1 Requirements

The library codes are not dependent on sources of any operating system. The only requirement is the GNU MP library for multiple precision arithmetic. It is a widespread library that can be installed on many systems. It is often a part of the software that is added to standard instalations of systems. If an operating system does not have additional software packages system or the system exists but it does not contain GMP, the library can be compiled from the source codes. More information about obtaining and installation of GMP can be found in [6] and [5].

3.1.2 Makefile

If the program make[1] is available then the library can be created by appended configuration file – makefile. The makefile is located in the folder with source codes of sympy–cpp. If the path to the GMP headers is not */usr/local/include/* or the path to the GMP library is not */usr/local/lib/*, the definitions of these paths need to be changed in the makefile. Library sympy–cpp can be constructed as a static library by `make static-lib` or a dynamic library by `make shared-lib`.

[1] It is a utility that builds executable programs and libraries from source codes automatically.

If an operating system does not have the program make, there is also the possibility of a common compilation of the source codes and linking of the GNU MP by a C++ compiler and linker.

The makefile also provides more possibilities for development and testing. The information about these features is located directly in the makefile.

3.1.3 License

The source codes of library sympy–cpp and the appended testing framework are under the terms of the New BSD License and GNU MP library is under Lesser General Public License version 3. Both licenses are placed in the folder with relevant source codes.

3.2 Quick and easy user's guide

There are various possibilities about how to use sympy–cpp in other source codes. The first way is the standard installation [Section 3.1]. In this case it is necessary to include `sympy-cpp.h` in user's codes and also the linkage of the program with sympy–cpp and GNU MP has to be configured.

The second possibility is the common compilation of sympy–cpp source codes and codes that use sympy–cpp. The requirement of GMP library is still in effect and the library has to be integrated into the compilation. This manner of usage is possible but it is not recommended. User's codes have to use header file *Expr.h* for basic work with the expressions. If new definitions of mathematical functions [Section 3.2.3] are required then files *Ex.h* and *Fx.h* need to be included too. Header file *functions.h* enables the mathematical functions that are implemented in sympy–cpp. This approach is complicated and it does not have any advantages. The first way is recommended.

The functionality of the library belongs to namespace `sympycpp` and therefore it is necessary to specify utilized elements of sympy–cpp by this namespace.

3.2.1 Basics

Class Expr represents mathematical expressions. Working with Expr is very intuitive and simple.

```
Expr a("a");
Expr b("b");
Expr c(31);
Expr d(2.756);
Expr e(1,"2345843908598538589589684096/23568348753753495839");
Expr f(2*a-b+Sin(a));
```

Figure 3.1: Construction of expressions by Expr's constructors.

Expr provides some constructors for the construction of expressions. There are simple constructors of variables, numbers and a copy constructor. All these

constructors are demonstrated on examples in [Figure 3.1]. Variables can be constructed by `std::string` and `char*` that specify names of constructed mathematical variables. Numbers are represented by Expr when its constructors are initialized by `int` and `double`. The construction of long rational numbers is provided by a constructor that has two parameters. The second parameter is the long number in text form and the first parameter is an `int` value that makes this constructor incompatible with the constructor of variables.

A very frequent method of construction is the copy constructor. More specifically, it is its application on combinations of mathematical operations, functions and other expressions. The provided operations are unary $(+, -)$ and binary $(+, -, *, /, \hat{})$. Operator $\hat{}$ expresses an exponentiation, but the priority of this operator is in C++ lower than in usual mathematics. Therefore it is better to use a lot of parentheses around the expressions or have the operator $\hat{}$ replaced by function `Power` [Figure 3.2].

```
Expr g(a*((2+a)^(3*b)));
Expr h(a*Power(2+a,3*b));
```

Figure 3.2: Equivalent representations of the expression $a(2+a)^{3b}$.

The group of implemented mathematical functions that can be used in construction of expressions is small but still expandable [Section 3.2.3].

Implemented also is an operator $=$. The construction of expressions by this operator is possible, but the copy constructor is faster. The operator should be used to assign any expressions into the already existing Expr objects [Figure 3.3].

Function `str` transforms stored expressions into `std::string` and thus presentations of stored information in Expr are performed by this function [Figure3.3].

```
Expr a("a"), b("b"), c("c"), g("g");
g = 2*a + c*Sin(pi/2) - a*b*(c-1);
std::cout << g.str() << std::endl;
g = 3*a/2 - Power(Ln(a), 2);
std::cout << g.str() << std::endl;
```

Figure 3.3: Assignment and printing of expressions.

3.2.2 Services

The main task of the library is the automatic simplification of expressions. This action is in operation during each construction and modification of expressions. It is not possible to switch off this simplification or implement it manually.

All other tasks are controlled by users. These are substitution, expansion, differentiation and Taylor series.

The substitution [Figure 3.4] implemented in sympy–cpp is a powerful symbolic manipulation. It manages actions ranging from simple substitution of variables to complex recursive replacing of expressions.

The substitution is performed by member function `sub(const Expr & exp1, const Expr & exp2)`, where the first parameter `exp1` is a replaced subexpression and the second `exp2` is a replacing expression. But a dangerous manipulation can

```
Expr x("x"), y("y"), z("z");
Expr a(x+y+z);
a.sub(x, y);
// x + y + z ⟶ 2y + z
Expr a(3*x+2*z-z*x+y/(y^x));
a.sub(x, 1);
// 3x + 2z − zx + y/y^x ⟶ 4 + z
Expr a(4*x*(x+y)*z);
a.sub(-2*x, y);
// 4xz(x + y) ⟶ −y^2 z
Expr a(4*(Cos(x-y)+(Sin(y-x)*(x-y)+z)*(y-x+z));
a.sub(x-y, pi/2);
// 4(cos(x − y) + ((sin(y − x)(x − y) + z)(y − x + z))) ⟶ 4(z − π/2)^2
Expr a(sin(1-(Power(x,-y)*((x+y)/z));
a.sub((x+y)/z, x^y);
// sin(1 − x^{−y} ∗ (x+y / z)) ⟶ 0
```

Figure 3.4: Examples of substitutions.

be created by the recursive simplification and substitution operating together. Sympy–cpp does not make any check of repeated cyclic substitution [Figure 3.5]. Prevention of these infinite loops is the responsibility of users. The simplest way of doing this is to use different variables in the replacing and replaced expressions.

```
Expr x("x"), e(2+x);
e.sub(3+x, x); // 2 + x ⟶ −1 + x ⟶ −4 + x ⟶ ...
```

Figure 3.5: Infinite recursive substitution.

Member function `expansion(const int level = -1)` expresses mutliplication of sums as summation of multiplication. This process is applied to a whole expanded expression if this `level` is negative. In other cases, expressions are expanded by levels. The level means a group of multiplications that belong to the same count of multiplications [Figure 3.6]. The number of the expanded levels is specified by parameter `level` [Figure 3.7].

$$\underbrace{x(2+\overbrace{y(a+\underbrace{b(2+x)}_{\text{third level}})(a-b)}^{\text{second level}})}_{\text{first level}} + \underbrace{(x+sin(cos(x^{\overbrace{2(a-x)}^{\text{second level}}})))^2(\overbrace{2(\underbrace{a(b+c)}_{\text{third level}}+x)}^{\text{second level}})}_{\text{first level}}$$

Figure 3.6: Levels of the expansion.

Differentiation of mathematical expressions has also a very simple usage [Figure 3.8]. Function `diff(const Expr & var)` makes differentiation with respect to `var` that has to be a variable[2].

[2]expression constructed by `Expr(const std::string)` or `Expr(const char *)`

```
Expr a("a"), b("b"), c("c"), d("d");
Expr e(a*(1-Power(b-c, 3*(a-d))+d*(2+a+3*(b-c))));
// ⟶ a(1 − (b − c)^{3(a−d)} + d(2 + a + 3(b − c)))
Expr e1(e), e2(e), e3(e);
e1.expansion(1);// ⟶ a − a(b − c)^{3(a−d)} + ad(2 + a + 3(b − c))
e2.expansion(2);// ⟶ a + 2ad + a^2 d + 3ad(b − c) − a(b − c)^{3a−3d}
e3.expansion(3);// ⟶ a + 2ad + 3abd − 3acd + a^2 d − a(b − c)^{3a−3d}
e.expansion();// ⟶ a + 2ad + 3abd − 3acd + a^2 d − a(b − c)^{3a−3d}
```

Figure 3.7: Examples of the expansion.

```
Expr a("a"), b("b"), c("c"), x("x");
Expr e(a*Sin(x/2)/x-(x^3)*(x-10));
e.diff(x);
//(a/x sin(x/2) − x^3(x − 10))' == −x^3 + 3x^2(10 − x) + a/(2x) cos(x/2) − a/x^2 sin(x/2)
e = Power((x+y),(2*x+3))*2*x;
e.diff(x);
//(2x(x + y)^{3+2x})' == 2((x + y)^{3+2x} + x(x + y)^{3+2x}(2ln(x + y) + (3+2x)/(x+y)))
e = 2*x*x*y*Ln(x)+(a^x);
e.diff(x);
//(a^x + 2x^2 yln(x))' == a^x ln(a) + 2(xy + 2xyln(x))
```

Figure 3.8: Examples of differentiation.

The last examined task of the sympy–cpp library is the Taylor series [Definition B.3]. The Taylor series is obtained as the return value of special member function `taylorSeries(const Expr & exp1, const Expr & exp2, const int n)`. Expressions that produce the Taylor series stay unmodified, this function may be therefore invoked by constant expressions.

```
Expr x("x");
e1 = Sin(x).taylorSeries(x, 0, 10));
//x − x^3/6 + x^5/120 − x^7/5040 + x^9/362880
e2 = Ln(1-x).taylorSeries(x, 0, 5));
// − x − x^2/2 − x^3/3 − x^4/4 − x^5/5
e3 = (Power(x, 3)*Cos(x)/(1-x)).taylorSeries(x, 0, 8));
//x^3 + x^4 + x^5/2 + x^6/2 + 13x^7/24 + 13x^8/24
```

Figure 3.9: Examples of Taylor series.

3.2.3 Mathematical functions

The sympy–cpp library includes a few mathematical functionsl, namely *sinus*, *cosinus*, *tangent*, *cotangent* and *natural logarithm*, which are applicable by functions `Sin, Cos, Tg, Cotg, Ln` [Figure 3.10].

```
Expr x("x"), y("y");
Expr e(Sin(x)+Cos(x));
e.diff(x);
Expr f(Ln(Cotg(x*y)-Tg(x));
f.sub(2+y, x);
```

Figure 3.10: Usage of predefined mathematical functions.

Fortunately, the deficient count of the functions is solved by user defined functions. There are two ways of doing this. Both methods require better understanding of sympy–cpp inner principles and the new mathematical functions are wrapped up by C++ functions that will carry out simplification.

The first concept [Figure 3.11] is very simple, it can be used with only a little information about the inner principles, but it has too restrictive properties. The construction is made by class Fx that has a few constructors [Section 4.1.7]. As can be seen in the example [Figure 3.11] of the Fx constructor, the first parameter is an argument of the constructed mathematical function and the second is the name of this function.

```
#include "sympy-cpp.h"
using namespace sympycpp;
Expr mySinus(const Expr & argument) {
    Fx function(argument, "Sinus");
    Expr expression(&function);
    return expression;
}
int main() {
    Expr x("x"), y("y");
    Expr e(2 + 4*x*mySinus(x+y) - x*mySinus((x+2*y-y)));
    e.diff(x);
    e.sub(y, x);
}
```

Figure 3.11: Example of user defined function by predefined class Fx.

In this case, the absence of exact differentiation is a problem. Differentiation is available only as iconic manipulation. The expression

$$2 + 3xSinus(x + y)$$

from the example [Figure 3.11], which uses added function `mySinus`, has differentiation

$$3(Sinus(x + y) + xSinus'(x + y)).$$

Taylor series of such added functions, that do not have the exact differentiation implemented are pointless. The other functionality works fine. Such adding of mathematical functions is sufficient, if exact differentiation and Taylor series will not be required.

The second concept [Figure 3.12] of adding the mathematical functions is based on deriving new classes from class Fx. The classes derived from Fx have to reimplement constructor `Fx(const Expr &)` and three functions: an exact differentiation as `virtual Ex * diff(const Sym &)`, an allocation of identical objects as `virtual Ex * copy()const` and a special construct function as `virtual Ex * create(Ex * arg, const allocationPolicy)const`. The last reimplemented function `create` creates new expressions from the first parameter by the particular wrapper construct function.

The wrapper functions present mathematical functions in users' source codes. These wrappers can carry out special simplification. The main feature of this simplification is the replacement of the mathematical functions by other, simpler expressions, if it is possible.

```
Expr Sinus(const Expr &);
class MySinus: public Fx {
public:
    MySinus(const Expr & e) : Fx(e, "Sinus") {}
    virtual Ex * diff(const Sym & x) const {
        Expr e1(Cos(x));
        Expr e2(e_->diff(x), STEALING);
        Expr e3(e1*e2);
        return e3.innerCopy();
    }
    virtual Ex * copy() const {
        return new MySinus(e_);
    }
    virtual Ex * create(Ex * arg, const allocationPolicy X) const {
        Ex * f = Sinus(arg).innerCopy();
        delete arg;
        arg = 0;
        return f;
    }
};
Expr Sinus(const Expr & e) {
    if (e.str() == "0") {
        return Expr(int(0));
    }
    Ex * args = new MySinus(e);
    return Expr(args, 1);
}
```

Figure 3.12: Example of user defined function by own class.

Programming of new mathematical functions only with possibilities of class Expr is awkward. The usage of inner sypmy-cpp methods makes the work more comfortable and efficient. The inclusion of the header file *sympy-cpp-develop.h* instead of *sympy-cpp.h* grants access to the whole inner functionality. Sufficient information about the inner part of sympy–cpp will be found in chapter 4, which should be read carefully before using any of the inner properties of the library.

Chapter 4

Inside sympy–cpp

This chapter is not a complete list of all constants, structures, classes and functions from the library. This list and description of its items can be found in the library documentation. The following text describes important classes, principles and algorithms.

Simplicity and lucidity are preferred aspects of the library, but a tardy library would not be really usable. Speed and simplicity often stand against each other. The library design tries to respect both requests in a reasonable ratio. Therefore some parts of the design can be obscure, but this section will make them more clear.

4.1 Representation of expressions

Mathematical expressions managed by the sympy–cpp library are represented as objects of class Expr. Expr is a wrapper class that covers all inner operations. The most essential role is played by class Ex and its descendants. Expr owns a pointer on Ex and provides an interface to call member functions of class Ex. Ex represents a general mathematical expression and its successors are representations of particular expressions.

It is obvious that there are two types of elementary expressions, namely variables and numbers. Complex expressions are created from other expressions by operations and functions. Each compound expression has special properties that are determined by the constructing operation, function.

For this reason, adepts for the classes which will represent particular expressions can be found among variables, numbers, operations and functions. Some classes can be formulated using other classes [Table 4.1].

operations	alternative representations	samples
unary plus, unary minus	encapsulated in every class	x and $-x$
subtraction	unary minus and addition	$x - y$ is $x + (-y)$
division	unary minus , multiplication and power	x/y is $x * y^{-1}$

Table 4.1: Alternative representations of particular mathematical expressions.

The representations of numbers, variables , additions, multiplications, powers

are classes Number, Sym, Add, Mul and Pow. A special case is the representation of the mathematical functions. Class Fx represents a general mathematical function and each mathematical function should be derived from Fx. One advantage of this design is the possibility to expand the basic set of mathematical functions by users' defined functions. The number of classes is relatively small but it does not cut down the flexibility of the library. The described relations among the classes are illustrated in the schema [Figure 4.1].

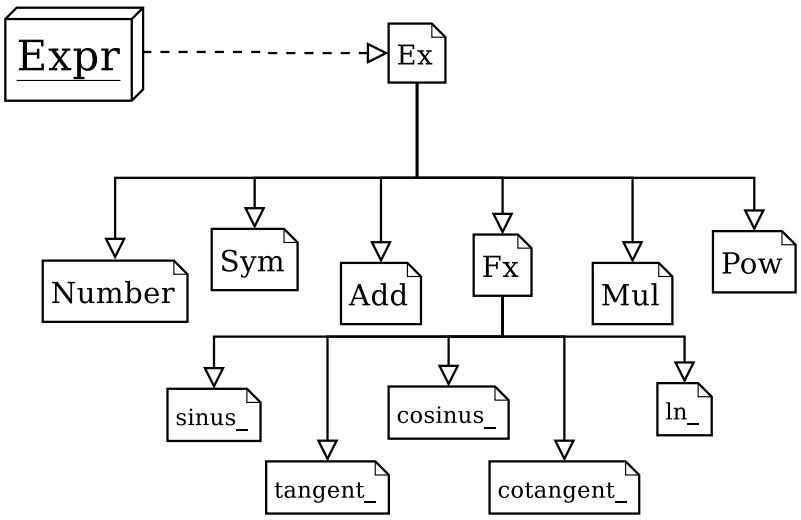


Figure 4.1: The relations of the classes that represent mathematical expressions. Arrows represent the relations: class B inherits from A ($A \longrightarrow B$) and class X owns pointer on Y ($X \dashrightarrow Y$).

4.1.1 class Ex

Ex represents an unknown mathematical expression. This unknown mathematical expression is an abstraction that can not be used in any computation as a stand–alone object, because it does not have enough information about the structures. The construction of Ex is acceptable only in its descendants, Ex being a component of them, while stand–alone objects of Ex are forbidden.

Ex provides only a few member functions that can return correct results from data available in Ex [Figure 4.3, 4.5] and other functions define interfaces [Figure 4.4] for specific expressions. The first group of functions is created from shared properties and expression type identifications. A unary sign is only one common property that arises from the elimination of some operations[1] [Table 4.1]. The type identification is not necessary but the knowledge of types permits more useful and also faster modifications than the invocation of virtual member functions. Defined interfaces determine the abilities of the whole library, because the front-end of the library does not induce any member functions of successors directly, but it uses the interfaces.

[1] unary plus, unary minus

Constructor

```
Ex(const type_info)
```

Figure 4.2: Constructor of Ex.

Management of signs

```
Sign sign() const
void sign(const Sign)
```

Identification of types

```
type_info type() const
bool isNum() const
bool isMul() const
bool isAdd() const
bool isSym() const
bool isPow() const
bool isFx() const
```

Figure 4.3: The list of Ex member functions that are implemented in Ex.

Interfaces

`virtual void treeView(const int i) const`
Transformation of expressions into suitable infix trees.

`virtual size_t rsize(const bool all = true) const`
Real size (rsize) is a number of elementary subexpressions.

`virtual size_t asize() const`
Actual size (asize) is a count of all immediate subexpressions.

`virtual size_t size() const`
Count of subexpressions.

`virtual std::string str() const`
Transformation of an expression into the std::string.

`virtual std::string unsigned_str() const`
Transformation of an expression absolute value into the std::string.

`virtual Ex * diff(const Sym &t) const`
Differentiation.

`virtual Ex * copy ()const`
Creation of a new identical expression (a clone).

Figure 4.4: The list of the interfaces that are defined in Ex.

4.1.2 class Number

This class represents numbers. It covers integers, rational and real numbers. Class Number is using an external library[2] for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers.

Class Number works in two modes. The first is a rational mode and the second is a real mode. Integers belong to the rational mode. Constructors [Figure 4.6] behave accordingly to this convention. The constructors parameters `N_Real, N_Rational, N_Real_init, N_Rational_init` are aliases for classes `mpf_class,`

[2]GNU MP [5]

Special identification of subtypes

(a)
```
virtual bool isMultiple() const
virtual Number Multiplicity () const
```

(b)
```
virtual bool isInteger() const
virtual bool isRational() const
virtual bool isReal() const
```

Figure 4.5: The list of Ex member functions that have partial implementation in Ex. This means that the information in Ex is not always sufficient for correct identification.

`mpq_class` in the external library and C++ types `double, int`. This class has automatic detection and conversion between the modes. Values of numbers are stored in `realVal_`[3] or `ratioVal_`[4] by the mode, and information about the active mode is owned by `isRatio_`[5].

Constructors

```
Number(const N_Real &)
Number(const N_Rational &)
Number(const N_Real_init)
Number(const N_Rational_init)
```

Figure 4.6: Constructors of class Number

Class Number implements almost all interfaces from class Ex, the common interface [Figure 4.4] and also the special number identification interface [Figure 4.5 (b)].

Setting-up of a new value

(a)
```
void setToAddition(const Number &, const Number &)
void setToSubtraction(const Number &, const Number &)
void setToMultiplication(const Number &, const Number &)
void setToDivision(const Number &, const Number &)
```
New stored value is the result of arithmetic operation.

(b)
```
void setValue(const Number &)
void setValue(const N_Rational_init)
void setValue(const N_Real_init)
void setValue(const N_Rational &)
void setValue(const N_Real &)
```
Simple setting up.

(c)
```
void setTFactorial(const int x)
```
New stored value is factorial of an ingoing parameter.

Figure 4.7: Setting-up of a new value.

[3] type `N_Real`
[4] type `N_Rational`
[5] type `Boolean`

Class Number also provides some member functions for convenient and secure work with numbers. The results of each arithmetic operation [Figure 4.7(a)] are tested after evaluation, and the mode is changed if necessary (since the addition of two real numbers can create integer).

Two comparing operands < and == [Figure 4.8] are implemented. The other comparing functions are not necessary, because they can be replaced by combinations of negations, logic conjunctions, implemented comparing functions and their parameter order.

Comparing

(a)
```
bool eq(const N_Real &) const
bool eq(const N_Rational &) const
bool eq(const N_Real_init) const
bool eq(const N_Rational_init) const
bool eq(const Number &) const
```

(b)
```
bool lt(const Number &) const
bool lt(const N_Real &) const
bool lt(const N_Rational &) const
bool lt(const N_Real_init) const
bool lt(const N_Rational_init) const
```

Figure 4.8: Comparative functions (a) equal (b) lower than.

The set of member functions is expanded by some useful support functions [Figure 4.9]. A function that is especially applicable is the conversion into `int` and also the control of possibility to execute this conversion.

Auxiliary functions

`Number abs() const`
Absolute value.

`std::string str2() const`
Alternative transformation to std::string, that does not use parentheses.

`bool isInt() const`
Is it possible convert Number into int?.

`int getInt() const`
Transformation to int.

`void checkVal()`
Verification of a Number value and a relevant mode.

Figure 4.9: Auxiliary functions.

4.1.3 class Sym

Variables are represented by class Sym. Sym implements the common interface [Figure 4.4] and adds some other functions. All variables need to have own an identifier. The identifiers make them distinguishable from each other. This means that any two objects of Sym are the same variables when they have

the same identifiers. The identifier of variables is stored in Sym as `std::string title_`. Only one constructor [Figure 4.10] is available by this identifier for class Sym.

Constructor

```
Sym(const std::string &)
```

Figure 4.10: Constructor of Sym.

The other functions [Figure 4.11] are operating with `title_`. The first one enables direct identification of Sym and the others provide convenient detection of the same variables by operators `==` and `!=`.

Identifier functions

```
std::string title() const
```

Return name/title/identifier of Sym/variable.

```
boolean operator==(cons Sym &) const
boolean operator!=(cons Sym &) const
```

Comparison of Sym by `title_` *(signs are ignored).*

Figure 4.11: Functions that are operating with `title_`.

4.1.4 class Add

The abstraction of addition is represented by the class Add. Addition is a binary commutative mathematical operation. Add exactly represents n-ary addition (where $n \geq 2$), which means that Add can be either an addition or a sequence of additions [Figure 4.12].

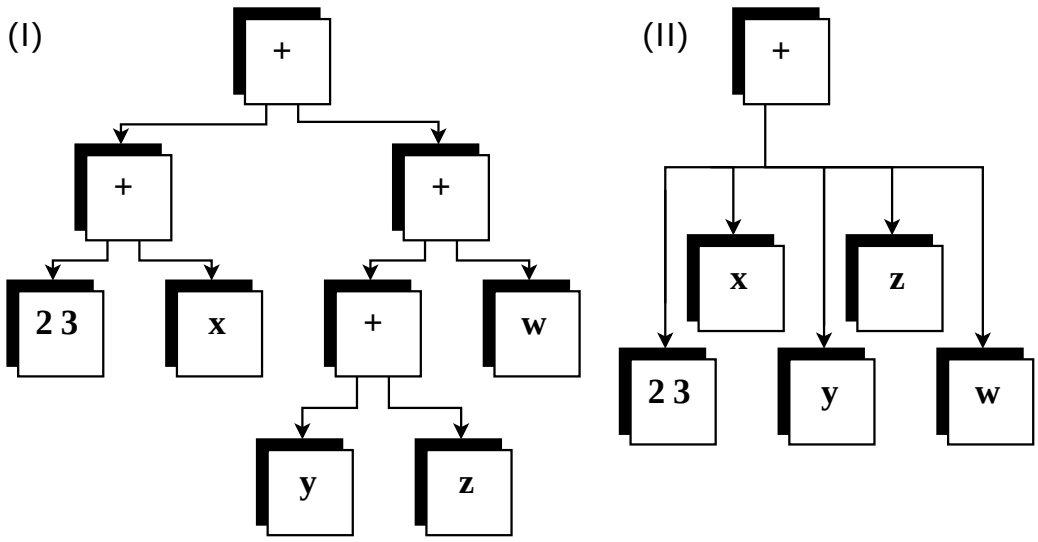


Figure 4.12: (I) usual tree representation of addition (II) shorter representation used in sympy–cpp

The possibility of having more sums in one object contributes to simpler and more efficient manipulation. The number of sums in sequence is changeable and class Add has to reflect this fact.

Addends are stored in a dynamic structure `basic_container`[6] that has to provide member functions begin()[7], end()[8] and forward_iterator. Each container that satisfies these conditions is able to replace an actually used container. There is one more requirement for the container, as it should have the possibility of being sorted by a supplied operator < [Section 4.4.6]. All containers implemented in STL are suitable.

The construction of addition from addends is managed by class Operations [Section 4.2.2], because a sum of some elements is sometimes reduced and thus it would not be an addition. This particular way of construction uses an unsuitable constructor that works with input `basic_container` [Figure 4.13]. The constructor assumes that the container is sorted[9]. The second parameter of this constructor is a flag of special memory management [Section 4.4.1]. The constructor exploits memory from the container for its own demands. Furthermore, the constructor does not check the validity of input data. It can create an invalid object and therefore individual usage may be potentially dangerous. The definition in the private section of class Add prevents the misleading application.

Copy constructor and virtual member function `copy` [Figure 4.13] are the last two possibilities in the construction of addition. They make a clone of any Add object in separate memory.

Construction of addition

`Add(basic_container &v, const allocationPolicy flag)`
Special private constructor.

`Add(const Add &)`

`virtual Ex * copy() const`
Copying.

Figure 4.13: Construction of Add.

Class Add further contains two added member functions [Figure 4.14],the first function is relevant in reusing the structures of Add. The second one provides information that is useful during construction and simplification of expressions [Section 4.2.2] like heuristics.

Added functionality

`bool omit(iterator & index)`
Omits one element from addition if possible (more than 2 addends).

`bool moreThan1Multiple() const`
Is there more than one multiple addends?

Figure 4.14: Add member functions.

4.1.5 class Mul

Class Mul is a representative of multiplications. Mul is very similar to class Add. The concatenation of series of multiplication, the storing of multiplicands

[6] defined in the file Ex.h

[7] returns an iterator to the beginning of the `basic_container`

[8] returns an iterator just past the last element of a `basic_container`

[9] [Section 4.2.1] sorting, ordering

in the container and the way of its construction use the same approaches as class Add in the previous section.

Naturally, it is possible to find some small differences between the two classes. Mul does not have the member function `moreThan1Multiple()` [Figure 4.14], which is pointless in this case. Some auxiliary functions have been appended [Figure 4.15] into the class.

Mul has a special policy for the management of multiplicands signs. Central sign administration is easier and faster than the distributed sign administration and therefore the sign of the whole multiplication mirrors the signs of all its multiplicands. The expression $-xy^2(-z)u(-w)$ is thus transformed to $-xy^2zuw$, where the sign does not belong to the first multiplicand, but it pertains to the whole multiplication.

Special cases of multiplication are multiple expressions that have only binary representation [Figure 4.16]. This is the only exception in the concatenation of series of multiplications. The multiplicity is the first multiplicand and the rest of the expression is the second one. Such representation is appropriate for more frequent manipulation with a multiplicity in contrast with other parts of multiplicative expressions.

Added functionality

`virtual Number Multiplicity() const`
Multiplicity of expressions.

`virtual bool isMultiple() const`
Is an expression multiple?

Figure 4.15: Mul member functions.

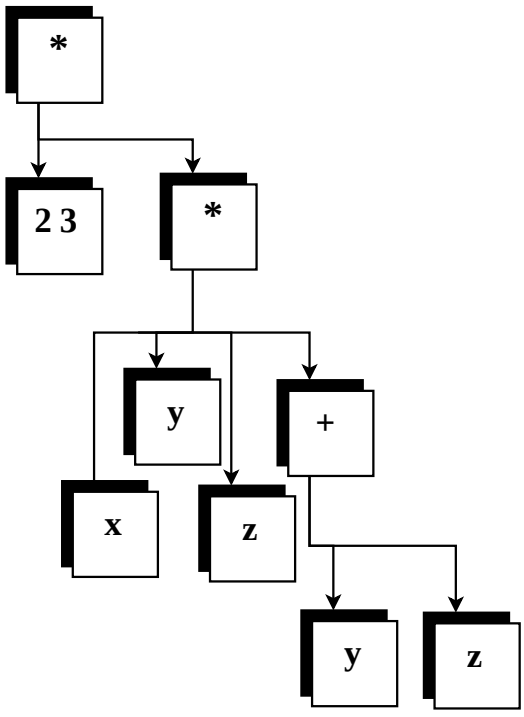


Figure 4.16: Multiple expressions always have the binary form. There is only one exception in the concatenation of multiplications.

Mul provides three private unsuitable constructors [Figure 4.17] and one public copy constructor. Each of the flawed constructors assumes sorted[10] multiplicands. The first constructor is an omitting copy constructor, which omits one prompted multiplicand. The other constructors exploit the memory from their input multiplicands. This behavior is specified by the last parameter of the constructor that marks special memory management [Section 4.4.1].

All these unsuitable constructors are not safe and they can create invalid objects, because ordering of the multiplicands is not tested and the amount of its operands is not controlled.

Construction of Mul

`Mul(const Mul &, const_iterator)`
Special constructor that omits one multiplicand.

`Mul(Ex *& v, Ex *& x, const allocationPolicy flag)`
Exploiting constructor.

`Mul(basic_container & v, const allocationPolicy flag)`
Exploiting constructor, which makes multiplication from elements of v.

Figure 4.17: Constructors of Mul.

4.1.6 class Pow

Pow represents powers. Powers consist of a base and an exponent, which are two independent mathematical expressions. The concatenation of powers is not necessary as it is already present in class Add and Mul. Series of powers are replaced by simple mathematical formula (4.1).

$$x^{a_1^{a_2^{\cdot^{\cdot^{a_n}}}}} = x^{a_1 a_2 \ldots a_n} \tag{4.1}$$

So the class does not need any dynamic container to store a variable count of subexpressions, because the mutable expression count is reloaded in Mul. Pow stores only two expressions for the base and exponent in member variables `base_` and `exponent_`.

Of course, copy constructor and member function `copy` are available and also there are the next two constructors [Figure 4.18]. The constructor which has `allocationPolicy` [Section 4.4.1] as its last parameter exploits the memory from input parameters and the second constructor manufactures copies. Pow provides function `copyInverted` [Figure 4.19a] which is a special alternative to member function `copy`. The difference between them is the sign of exponents, because function `copyInverted` changes the sign. This is useful when an inversion of a fraction is required.

The group of functions [Figure 4.19b] procures some actions that are applied to exponents. These are simple detections of an integer in exponents and an exponent sign. The last action is the conversion of exponent absolute values into the `std::string`.

[10] for more information about the sorting see section 4.2.1

Construction of powers

```
Pow(const Ex * b, const Ex * e, const allocationPolicy)
Pow(Ex * &, Ex * &)
```

Figure 4.18: Construction of powers.

Function `ganef` [Figure 4.19c] deserves special attention, because it has very strange application. The parameters of the function need to be empty before stepping in. The function moves the base and exponent of the power into these parameters and then the object of Pow is not a valid power, thus signifying that this object should be deleted. Function `ganef` is useful when base or exponent can be reused in new expressions and power as an entity is redundant.

Auxiliary member functions

(a) `Ex * copyInverted() const`
Construction of copies that have changed the sign of the exponents.

(b) `bool isExponentInteger() const`
`bool isExponentPositive() const`
`std::string abs_Exp_str() const`
Functions of the exponent.

(c) `void ganef(Ex * &, Ex * &, const allocationPolicy)`
Exploiting of base and exponent.

Figure 4.19: Special member functions of Pow.

4.1.7 class Fx

Class Fx represents general mathematical functions. All mathematical functions in the library should have connection with this class. Predefined sympy–cpp mathematical functions are constructed by classes derived from Fx and also functions that are added by users should use class Fx or classes derived from Fx. More information on using predefined functions and creating new mathematical functions can be found in section 3.2.3.

Functions have names and arguments. Names of functions are stored in `std::string name_` and expressions that represent the arguments are stored in `ex * e_`. Construction is made by some constructors [Figure 4.20], function `copy()` and special construction member function `create` [Figure 4.21].

4.1.8 class Expr

Class Expr covers and unifies pointer arithmetic of classes that represent expressions. This class is intended for standard work with expressions. Constructors [Figure 4.22] and member functions [Figure 4.23] provide full functionality of the library.

Description of Expr usage and some simple examples of creating and handling of expressions by Expr are presented in section 3.2.

Construction of Fx

`Fx(const Fx &)`
Standard copy constructor.

`Fx(const Ex *, const std::string &)`
`Fx(const Expr &, const std::string &)`
Construction of functions by arguments and names.

`Fx(Ex * &, const std::string &, const allocationPolicy)`
Constructor by argument and name, but memory from input expression is exploited for the argument.

Figure 4.20: Constructors of Fx.

Member functions of Fx

`virtual Ex * copy() const`
Construction of new identical objects in separate memory.

`virtual Ex* create(Ex * & e, const allocationPolicy) const`
Construction of new functions.

`std::string name() const`
Name of the represented function.

Figure 4.21: Member functions of Fx.

Constructors of Expr

`Expr(const Expr &)`
Standard copy constructor.

`Expr(const Ex *)`
`Expr(Ex * &, const allocationPolicy)`
Wrapping of pointers. (the first one makes a copy, the second constructor exploits ingoing memory)

`Expr(const std::string &)`
`Expr(const char *)`
Construction of elementary expressions (variables)

`Expr(const int)`
`Expr(const double)`
`Expr(const int, const char *)`
Construction of elementary expressions (numbers). The last constructor creates rational numbers from text strings (the first parameter is only a mark that makes this constructor distinguishable from the constructors of variables).

Figure 4.22: Constructors of Expr.

Expr also provides some auxiliary member functions [Figure 4.24]. These functions are important for the implementation of new mathematical functions. They permit direct sign management and return pointers on hidden representations of expressions. The pointers allow usage of inner functions, that have better performance and more abilities. The "pointer" functions often manipulate with the memory and furthermore, the user is able delete the memory manually. This is dangerous for the validity of Expr objects and therefore the pointers, that are returned by function `innerCopy` point to copies of hidden parts of the original

Functionality of sympy–cpp

`bool expansion(const int level=-1)`
Transformation of multiplication of sums to summation of multiplication.

`void sub(const Expr & x, const Expr & y)`
Substitution of the first parameter `x` *by the second parameter* `y`.

`Expr taylorSeries(const Expr & x, const Expr & a, const int n) const`
It returns elements, up to `n`*-th differentiation, from Taylor series by* `x` *in point* `a`.

`Expr & diff(const Expr & x)`
Differentiation with respect to `x`.

Figure 4.23: Member function of Expr.

objects. However, this is not safe enough, because automatic memory release is not present and thus users need to do it manually.

Auxiliary functions

`Ex * innerCopy()const`
Copying of expressions.

`std::string tree()const`
Transformation of expressions into std::string, that shows expressions as n-ary trees.

`void sign(const Sign &)`
`Sign sign()const`
Sign management.

Figure 4.24: Member functions of Expr.

4.2 Construction and simplification of expressions

Construction and simplification of expressions are very closely linked processes that are heavily dependent on recognitions of similarities in expressions. The identification of expression types by member functions [Figure 4.3] is not satisfactory. Consequently, sympy-cpp defines ordering of the mathematical expressions as an alternative classification.

4.2.1 Ordering – class CompararIx

Ordering is managed by class ComparerIx, which contains functions for comparing expressions [Figure 4.25] and detecting compatibility for simplification [Figure 4.26]. The main role of the comparative functions is during creation of new expressions, because expressions that can not be simplified are sorted by it.

The objective of ordering is faster search of particular operands in n-ary operations. Detailed description of the expression construction by merging is provided in section 4.2.2.

Comparison

`static bool addLessThan(const Ex * L, const Ex * R,`
`const bool compareSignSym = true)`
Special comparison for elements of additions.
`static bool mulLessThan(const Ex * L, const Ex * R,`
`const bool compareSignSym = true)`
Special comparison for elements of multiplications.
`static bool auxLessThan(const Ex * L, const Ex * R,`
`const bool compareSignSym = true)`
General comparison.

Figure 4.25: Special comparison functions.

The common comparative function for classes that represent addition and multiplication needs some corrections of compared expressions, because their simplification rules are different. Consequently, there are functions `addLessThan` and `mulLessThan`, which regulate some specific cases and then common comparative function `auxLessThan` is invoked. Each comparative function [Figure 4.25] has a special last parameter, whose default value is `true`. This parameter denotes whether signs of variables (`class Sym`) are included in the comparison (default choice) or not (setup `false`).

Algorithm: comparison of expressions

1. The transformation of expressions by provider operations.
   (a) addition
   If an addend is multiple, then its multiplicity is removed and the rest of the expression, without the removed multiplicity, is compared. This transformation is applied on the whole addend, but not on its subexpressions.
   Example: $6x(3-4y) \longrightarrow x(3-4y)$
   (b) multiplication
   If a multiplicand is a power, then its exponent is removed and its base is compared. This transformation is applied on the whole multiplicand, but not on its subexpressions.
   Example: $(x+y^2)^3 \longrightarrow x+y^2$
2. Comparing the counts of elementary subexpressions (numbers and variables). If the counts are equal, the comparing continues to the next step.
3. Comparison of expression types. The order of types is defined as: $numbers < variables < powers < additions < multiplications < functions$. If the expressions are of the same type the last step follows.
4. (a) compound expressions
   If the expressions are compound, the comparative mechanism is applied on their subexpressions up to the point of encountering first expressions that is not equal. If there are no differences between them, the compared expressions are equal.

(b) plain expressions
Numbers are ordered by the usual operator < for numbers and variables are compared by text comparison that is applied on their identificators.

Compatibility

```
static compatibility compatibilityForAddition(const Ex * L,
const Ex * R)
```
Is it possible to simplify the sum of input expressions?
```
static compatibility compatibilityForMultiplication(
const Ex * L, const Ex *  R)
```
Is it possible to simplify the multiplication of input expressions?

Figure 4.26: Detection of compatible expressions for simplification.

The second task of ComparerIx is checking for the compatibility of expressions for simplification. These functions [Figure 4.26] identify relations between expressions and return information about them. The information is stored in enumeration `compatibility` [Figure 4.34].

The compatibility control uses the same mechanisms as comparing. The first step is the cutoff of multiples/exponents and then the reduced expressions are compared as text.

4.2.2 Simplification & construction – class Operations

The construction of expressions is performed by class Operations. Services of Operations are used by operators [Figure 4.27], which are applicable in user's codes. The function Power, which is an alternative to `operator^` is also placed here (for more informations see [Section 4.1.6]).

Operators

```
Expr operator+(const Expr &, const Expr &)
Expr operator-(const Expr &, const Expr &)
Expr operator*(const Expr &, const Expr &)
Expr operator^(const Expr &, const Expr &)
Expr operator/(const Expr &, const Expr &)
Expr Power(const Expr &, const Expr &)
```

Figure 4.27: Operators on expressions.

These operators call particular static member functions from class Operations [Figure 4.28] that are responsible for customization of required forms of expressions to special forms. Expressions in the special forms are expressed only by addition, multiplication and powers [Table 4.1].

Class Operations provides alternative member functions [Figure 4.29&4.30], which work directly with the inner structures of expressions and therefore they have better performance than "operator" functions [Figure 4.28].

Finally, functions `addition, multiplication` and `power` have basic positions in the simplification and construction of expressions, because only opera-

Member functions of Operations

```
static Expr operatorAdd(const Expr &, const Expr &)
static Expr operatorSub(const Expr &, const Expr &)
static Expr operatorMul(const Expr &, const Expr &)
static Expr operatorDiv(const Expr &, const Expr &)
static Expr operatorPow(const Expr &, const Expr &)
```

Figure 4.28: Functions that transform some unsupported forms of expressions to the supported forms.

Member functions of Operations

```
static Ex * addition(const Ex *, const Ex *)
static Ex * subtraction(const Ex *, const Ex *)
static Ex * multiplication(const Ex *, const Ex *)
static Ex * division(const Ex *, const Ex *)
static Ex * power(const Ex *, const Ex *)
```

Figure 4.29: Functions, which have the same tasks as the functions from Figure 4.28, but these functions manipulate directly with the inner representations.

Member functions of Operations

```
static Ex * addition(Ex * &, Ex * &, const allocationPolicy)
static Ex * subtraction(Ex * &, Ex * &, const
allocationPolicy)
static Ex * multiplication(Ex * &, Ex * &, const
allocationPolicy)
static Ex * division(Ex * &, Ex * &, const allocationPolicy)
static Ex * power(Ex * &, Ex * &, const allocationPolicy)
```

Figure 4.30: The same as Figure 4.29 with a reuse of input memory.

tions addition, multiplication and exponentiation are represented by classes (Add, Mul and Pow). Consequently, only these operations can be simplified.

The simplification of Pow is simpler than the simplification other classes. It is managed by few rules [Figure 4.31] that check bases and exponents of constructed powers.

The simplifications of Add and Mul are realized by the same algorithm. This algorithm is based on the ordering of expressions [Section 4.2.1], merging of sorted sequences of expressions and identification of similarities in expressions.

Algorithm: simplification of Add and Mul

1. Construction of two sequences whose elements are operands of constructed operations.

   (a) If the type of the operand is different from the type of the constructed expression, this operand is inserted into the sequence. Such constructed sequence has only one this element.

   (b) If an operand and a constructed expression have the same types, operands of the operand are inserted into the sequence.

Simplification of powers

Let A,B and C be expressions.

- $A^0 \longrightarrow 1$
- $A^1 \longrightarrow A$
- $1^A \longrightarrow 1$
- $0^A \longrightarrow 0$
- $number^{integer} \longrightarrow number$
- $number^{-integer} \longrightarrow \frac{1}{number}$
- $(-A)^B \longrightarrow (-1)^B A^B$
- $(A)^{B^C} \longrightarrow A^{BC}$
- $(AB)^C \longrightarrow A^C B^C$

Figure 4.31: The simplification rules for exponentiation.

2. The sequences are compared step by step and merged into a resulting sequence.
   (a) If the elements are compatible [Figure 4.26], they can be joined to create a simpler new form. This joined form will contribute to the result and the original elements are deleted.
   (b) If the elements can not simplify each other, the lower one as defined by ordering [Figure 4.25] is moved into the result and the higher one is compared with the next element of the second sequence.
3. Construction of expression.
   (a) If the resulting sequence is empty, the expression is 0.
   (b) If the resulting sequence has only one element, the constructed expression is this element.
   (c) In all other cases, the expression is constructed from the resulting sequence by particular operation/expression constructor.

This algorithm has to warrant for the construction of valid objects. There is only one place (step 2a) that could infract the ordering. Fortunately, the construction of a simpler form is managed by rules [Figure 4.32&4.33] that are designed to keep the ordering. Simplifying expressions can eliminate each other or their multiplicity can be changed (simplification in addition) or modifications can be done only in the exponents of powers (simplification in multiplication). Such manipulations do not change the ordering.

4.3 Functionality

The abilities of the library should be provided as member functions of the classes that represent expressions. But some manipulations reorganize the structures of expression too much, so in these cases the solutions by other classes that reconstruct or build new expressions are prefered.

Simplification of addition

Let A,B and C be expressions and x, y, z numbers.

- $A + A \longrightarrow 2A$
- $xA + yA \longrightarrow zA,\ where\ z = x + y$
- $A + 0\ or\ 0 + A \longrightarrow A$

Figure 4.32: The simplification rules of addition.

Simplification of multiplication

Let A,B and C be expressions and x,y,z numbers.

- $1 * A\ or\ A * 1 \longrightarrow A$
- $0 * A\ or\ A * 0 \longrightarrow 0$
- $-1 * A\ or\ A * (-1) \longrightarrow -A$
- $AA \longrightarrow A^2$
- $A^B A^C \longrightarrow A^{B+C}$

Figure 4.33: The simplification rules of multiplication.

4.3.1 Expansion

This manipulation makes fundamental changes in structures of expressions. Class Utilities contains member function `expansion(Ex * & e, const int level)` that transforms multiplication of summation to summation of multiplication in expression `e`. The parameter `level` defines the number of expanded levels of multiplication. The description of the levels of multiplications can be found in section 3.2.2.

The algorithm is based on Depth First Search (DFS) and the count of found multiplication (levels of multiplication). DFS finds and counts levels of multiplication. If a branch of a tree representation is examined or a required level is achieved then the expansion is applied on the completely processed multiplication and DFS continues in the ensuing branches. This approach ensures the expansion of multiplication with already expanded subexpressions (up to defined level). The last expanded node is the root of the tree.

Special cases are powers that have an integer in their exponents and bases are sums. Simple sums of two elements can be expanded by Binomial theorem (BT) [B.1] and sums with more addends are expandable by Multinomial theorem (MT) [B.2]. Class Utilities provides function `multinomialTheorem(const Add &, const Number &)` that implements MT. Function that would implement BT is not yet available, but `multinomialTheorem` is able to compensate it. The implementation of BT remains important, as it going to increase the speed of expansion.

4.3.2 Substitution

Substitution is performed by class Substitution. The replacing of expressions by expressions is managed by member function `sub(Ex * & originalEx, const`

`Ex * fromEx, const Ex * toEx)`, which uses special substitution functions provided by this class.

The algorithm of substitution is based on Breadth First Search (BFS) and simple Linear Search (LS). Comparative functions from class CompareIx [Section 4.2.1] are often used here. The search for replaced expression by BFS is primarily performed in the subtrees. After that, equivalent nodes are searched by LS.

If the replacement is done, the process of substitution has to start again from the beginning again, because the original expression could have been transformed by automatic simplifications and consequently new occurrences of subexpressions that are eligible for replacement could have been created. The process of cyclic substitution is ended when no more substitution is applied.

4.3.3 Differentiation

Differentiation is a member function of the classes that represent the particular expressions. The differentiations are determined by a table of basic differentiations and simple rules [Section 2.3.3].

The only complex case of differentiation occurs when an expression (power) is differentiated with respect to a variable and this variable is located in its base and exponent. Let this expression be $f(x)^{g(x)}$ then its differentiation

$$\left(f(x)^{g(x)}\right)' = f(x)^{g(x)}\left(g'(x)Ln\left(f(x)\right) + g(x)\frac{f'(x)}{f(x)}\right).$$

This modification is eventuated from equality

$$\left(f(x)^{g(x)}\right)' = \left(e^{Ln\left(f(x)^{g(x)}\right)}\right)' = \left(e^{g(x)Ln(f(x))}\right)'.$$

The derived formula enables the solution of this differentiation by the already mentioned techniques.

4.3.4 Taylor series

Taylor series can be obtained from class Utilities by function `taylor(const Ex * fun, Ex * & tay, const Sym * x, const Ex * a, const int n)`. This function assigns elements from the Taylor series of the mathematical function `fun` to point `a` and into expression `tay`. The number of the assigned elements is implied by parameter `n` that fixes maximal allowed differentiation with respect to `x`.

The implementation uses already mentioned functions differentiation and substitution. Construction of these series is iterative by formula [B.3].

4.4 Auxiliary structures

4.4.1 allocationPolicy

Enumeration `allocationPolicy` has only one element `STEALING`. Its purpose is to label the functions that do not have any traditional memory management. Sympy–cpp does not implement any memory manager and therefore the memory

is frequently wasted. Allocation and release of memory decrease the speed of computation. To solve this problem, there are alternative functions that can reuse input memory. The Labeling of these functions is required to distinguish them from the standard functions. Marked functions have the last redundant parameter of the `allocationPolicy` type.

4.4.2 Sign

The type `Sign` is renamed to C++ type `bool`. It represents the unary signs of expressions. The two constants (macros) are redefined accordingly: `true` as `plus` and `false` as `minus`.

This improves legibility, lucidity and comprehensibility of the source codes in the library.

4.4.3 compatibility

Construction of expressions detects possible simplification and performs them. These two processes are separated and communication is ensured by enumeration `compatibility`.

It contains nine enumerators. The first three enumerators are common, the next three belong to the construction of addition and the last three are intended for the construction of multiplication.

`enum` {
`INCOMPATIBLE`, *//Relationship is not detected.*
`NUMBER_NUMBER`, *//Both compared expressions are numbers.*
`PLAIN_PLAIN`, *//Expressions are compatible and neither is multiple.*
`MULTIPLE_MULTIPLE`, *//Expressions are multiple.*
`PLAIN_MULTIPLE`, *//The second expression is a multiple of the first.*
`MULTIPLE_PLAIN`, *//The first expression is a multiple of the second.*
`BASE_BASE`, *//Both expressions are powers and their bases are compatible.*
`BASE_PLAIN`, *//The first expression is a power and its base is compatible with the second expression.*
`PLAIN_BASE`, *//The first expression is compatible with the base of the second expression that is a power.*
} `compatibility;`

Figure 4.34: Information about compatibility of expressions for simplification.

4.4.4 type_info

The type `type_info` is designed for identification of expression types that are the same as types of classes that represent particular expressions.

4.4.5 Numbers

There are two aliases for number types from GNU MP and two aliases for standard C++ number types. `N_Real` and `N_Ratio` are long number classes `mpf_class` and

```
enum {
EX, //general expressions – class Ex
NUM, //numbers – class Number
SYM, //variables – class Sym
POW, //powers – class Pow
ADD, //addition – class Add
MUL, //multiplications – class Mul
FX, //functions – class Fx
} type_info;
```

Figure 4.35: Identification of expression types.

`mpq_class`, which are used for storing numbers in sympy–cpp class Number. C++ types `double` and `int` are initial types of `N_Real` and `N_Ratio`. Consequently, these types have aliases `N_Real_init` and `N_Ratio_init`.

4.4.6 basic_container

The type `basic_container` is an alias for a container that is used for storing operands in classes Add and Mul. At the moment there is no container implemented in sympy–cpp and therefore the library utilizes a container from STL. Minimal requirements on the container are an implementation of forward iterators and function `containerSort`. This function is a wrapper of the sorting that is applied on the container [Figure 4.36]. Functions `ComparerIx::addlt` and `ComparerIx::mullt` are used as an operator $<$ in the sorting.

```
namespace sympycpp {
    void containerSort(std::list<Ex *> & l,
                       bool (* lessThan)(const Ex *, const Ex *)) {
       l.sort(lessThan);
    }
    void containerSort(std::vector<Ex *> & v,
                       bool (* lessThan)(const Ex *, const Ex *)) {
       sort(v.begin(), v.end(), lessThan);
    }
}
```

Figure 4.36: Examples of wrapper function `containerSort`.

The change of this container is a very simple act. It needs to change of `basic_container` definition in file *Ex.h* and also implementation of function `containerSort`. Containers `std::vector` and `std::list` have these wrappers already implemented.

4.4.7 Exceptions

Only two exceptions occur in Sympy–cpp because its detection of failures is not finalized enough. There are the `NotInt` and `ZeroDivision` exceptions. `NotInt` is thrown when an expression is not a convertible number with standard `int` and

the conversion is required. The second exception occurs when a denominator of a fraction is equal to zero. Both exceptions are derived from `std::exception`.

4.4.8 Utilities

Class Utilities provides diverse functionality. A big part of the already mentioned functionality belongs to this class. Taylor series, expansion and Multinomial theorem are among its members. The class also implements functions `void set0_2Pi (Ex *, bool &)` and `Expr set0_2Pi(const Expr &, bool &)` that transform expressions into interval $< 0, 2\pi)$, if this is possible. This is very useful for simplifications of goniometric functions.

4.4.9 Constants

The constants in sympy–cpp do not have a special implementation. There is a chance to define them as constant variables. This way is used for the definition of the one and only constant π, that definition being `const Expr::Expr pi("pi")`.

4.4.10 Mathematical functions

Mathematical functions that are provided by sympy–cpp are only samples of functional expanding. The implemented functions `Sin, Cos, Tg, Cotg` and `Ln` are simply replaceable by others [Section 3.2.3].

4.4.11 TestX

Class TestX is not a part that executes symbolic manipulation. It is a test framework that shall improve the work of developers. The tests have four levels of reports [Figure 4.37] and they can show a preview of the tree representation of expression. These properties are configurable by constructor [Figure 4.38a] and two member functions [Figure 4.38b].

The main role is in test function [Figure 4.38c]. The first parameter is the tested expression and the second one is an anticipated textual form of this expression. The third parameter is a line where the expression is located. It should be configured by macro `__LINE__`. And the last parameter is a test number that determinates the tested expression in a series of tested expressions. It enables testing of only one incorrect expression.

The test framework provides a series of tests. There are some tests in files *Tdifferentiation.cpp, Tsubstitution.cpp* and *Tpower.cpp* that should be compiled [Section 3.1.2] after each alternation of the library source codes.

Length of reports

short report
The report consists of the number of the test and the number of the line in a file where the test is located.

long report
An actual textual form of the tested expression and a required textual form of this expression is added to the short report.

Levels of reports

level 0
Correct expressions do not produce any message and incorrect expressions produce the long report.

level 1
Both cases produce the short reports.

level 2
Correct expressions produce the short form, incorrect expressions produce the long form of the report.

level 3
Both cases produce the long reports.

Figure 4.37: Reports and levels of reports in TestX.

Abilities of TestX

(a) `TestX(const int voice = 3, const bool tree = false)`
Constructor that enables the setup a level of reports (voice) and a possibility of the preview of the tree representation.

(b) `void setTree(const bool)`
`bool setVoice(const int)`
Setup of the preview of the tree representation and levels of reports.

(c) `void test(const Expr & e, const std::string eStr, const int line = -1, const int nT = 0)`
Test function.

Figure 4.38: Abilities of TestX.

Chapter 5

What is next

The library is operational but here is still a big room for improvement, as some functionality is still missing and some traits should have been done in another way.

5.1 Supplementation

A number of theoretical methods was not implemented and some technical problems decrease the quality of the library.

The technical problem of the memory management is probably the most considerable. Frequent allocations and deallocations of objects decrease performance of the library. A manager that would handle these routines can inhibit this shortcoming.

At the present time the Binomial theorem is managed by a function that is designed for Multinomial theorem. However, the Binomial theorem is a special case of Multinomial theorem and therefore it could be implemented by more sophisticated methods.

Elimination and reduction of expressions during the automatic simplification do not have any memory of their conditions. Unperformed supervision of conditions can creates failures in upper functions, which do not check and neither have the possibility of checking these conditions. An example of this mistake is in Wolfram Mathematica [Figure 5.1].

Bug in Wolfram Mathematica 7

```
In[5]:= Solve[((x - 1)*(x^2 - 3*x + 2))/(x - 1) == 0, x]
Out[5]= {{x -> 1}, {x -> 2}}
```

Figure 5.1: The bug in Wolfram Mathematica 7 for students version 7.01.0 for Linux. The equation $\frac{(x-1)(x^2-3x+2)}{x-1}$ has only one solution $x = 2$, because $x = 1$ does not belong to the domain of the equation.

The addition of missing functionality is also required as the next step in the development of sympy-cpp. Integration, limitation, progressive simplification and factorization should be added into the core of the system.

5.2 Improvements

Sympy-cpp has, like any other program, numerous areas in need of improvement. A special care is required for numbers. The sympy–cpp library uses the library GNU MP. More specifically, it is the usage of its C++ wrapper classes of C routines. These wrappers are slower than the C routines. The first improvement would be a direct application of those routines.

The handling of exceptions in sympy–cpp is neglected, but this topic is closely linked with the storage of the memory of conditions that is still not implemented.

The functions that provide substitution have to be redesigned and recoded because they are inefficient. These functions are too general and they do not use all information about the substituted expressions. For example let 0 substitutes x in $2xyzw\ldots$, which represents a very complicated and long multiplicative expression. The result of this substitution is identifiable from the second element in the expression, because $0 * anything = 0$, but the actual substitution functions do not detect this case. Consequently, the substitution is very slow.

The group of the mathematical functions have to be extended because the actual state is not satisfactory. This problem can be solved by users' defined functions, but this is not the most effective way.

Chapter 6

Summary

The Wolfarm Mathematica, Maple, GiNaC, Giac, Sympy, Maxima, Sage, Yacas are symbolic manipulation programs or libraries with big differences between each other. There are programs that have only one function – to make symbolic manipulation. Also, there are applications that have a wider functionality. Whether it is a simple little library or a huge mathematical system, they should have common goals. Results have to be correct and computation should be ended in a reasonable time.

The second requirement is not so easy to accomplish. The commercial programs as Mathematica and Maple are traditionally very fast in computation, but the group of free programs do not often have the sufficient performance. Despite this fact, there are libraries GiNaC and Giac, which are very fast, but their design is too complicated. Such libraries are not practically modifiable or the programmer must spend a lot of time to make these modifications. These libraries belong to the open sources software, but the already mentioned problem makes it inconsequential. On the other hand, they are very useful as a background for complex computations in some programs. The other programs are slower than those libraries or commercial programs.

6.1 Benchmarks

The aim of the developed library was to keep the design easily comprehensible, expandable and modifiable. The simplicity of the project is accomplished. The number of classes that are implemented in sympy–cpp is not too extensive, but it is sufficient to express mathematical expressions and operations with them. The classes are not too big and unclear, although there is some long member functions. Still, even these functions have relative simple structures.

The question that remains is whether this design is usable in a real life. Consequently some comparative tests have been added to the thesis.

These benchmarks should demonstrate that sympy–cpp is useful. Programs that have been included into the tests are Wolfram Mathematica 7 for students (Linux version) [2], GiNaC 1.4 [3], Sympy 0.5.15 [1], Sage 3.4 [4] and sympy–cpp. The first of the mentioned programs is the only commercial software in the tests. It is a fairly expensive application. Despite this fact, it is a very popular tool that is used very often. The second tool is a very fast library, that demonstrates the speed of open source software. Sympy is delegate of simple design and Sage is an

open source alternative to huge mathematical systems as Mathematica, Maple, Magma and Matlab. This diversified sample of symbolic packages covers a wide field of usage.

The benchmarks show only nominal features of those systems. Consequently, these tests do not determine exact performance of the programs, but it is a check whether sympy–cpp could step in the game.

All tests are performed fifteen times and average values are computed by statistical software R.

6.1.1 Expansion

The expansion has been tested in eight tests, that are mainly aimed on Multinomial theorem.

test 1
$(a+b+c)^{100}$

test 2
$(a+b+c)^{278}$

test 3
$(a+b+c)^{477}$

test 4
$(a+a(b-c)+c)^{73}$

test 5
$(a+a(b-c)a(b+c)+c)^{83}$

test 6
$(a+ab-ac+c)^{77}$

test 7
$(a+a^2b^2-2a^2bc+a^2c^2+c)^{37}$

test 8
$(a+b+c+d+e+f+g)^{15}$

program	t1	t2	t3	t4	t5	t6	t7	t8
Mathematica	0.45	2.87	10.65	3.11	4.65	3.58	1.69	2.19
GiNaC	0.08	1.3	6.19	1.89	3.3	3.28	3.29	0.63
sympy–cpp	0.5	4.22	13.14	9.88	14.71	11.56	3.93	6.87
Sympy	15.06	∞	∞	∞	∞	∞	∞	∞
Sage	12.6	∞	∞	∞	∞	∞	∞	∞

Table 6.1: ∞ means time longer than 30s

The tests show that sympy–cpp has a good starting position. Algorithms that are used in sympy–cpp have simple design. Optimization of these algorithms could increase performance and thus it puts sympy–cpp near to the top.

6.1.2 Taylor series

The programs were tested in five tests. These tests observe more features of the symbolic packages, because the construction of Taylor series needs more functionality. Differentiation, substitution and construction of expressions were used.

Sympy–cpp has a performance problem with substitution in long complex expressions. The substitution is correct but really slow. This problem was pinpointed as the priority for redesigning and recoding. The test samples try to minimize complex substitution.

test 1
Taylor series of $sin(a)$ in point 0 up to 100-th derivative by variable a.

test 2
Taylor series of $sin(a)$ in point $b-2$ up to 100-th derivative by variable a.

test 3
Taylor series of $sin(a)$ in point 0 up to 5000-th derivative by variable a.

test 4
Taylor series of $\frac{a^x}{cos(a)}$ in point 2 up to 7-th derivative by variable a.

test 5
Taylor series of $sin(cos(a^3))$ in point 0 up to 7-th derivative by variable a.

program	t1	t2	t3	t4	t5
Mathematica	0.28	0.32	9.95	0.34	0.27
GiNaC	0	0	0.14	0	0.01
sympy–cpp	0	0.01	0.38	3.37	3.47
Sympy	0.66	—	∞	—	2.47
Sage	1.46	1.55	∞	0.34	1.5

Table 6.2: ∞ means time longer than 30s, — means missing functionality

The tests of Taylor series demonstrate that sympy–cpp has fairly good performance. The tests 1,2 and 3 confirm this fact. The last two tests are highly affected by the slow substitution.

Chapter 7

Conclusion

The commercial symbolic manipulation packages are expensive and their source codes are closed. The mass of open source software is relatively slow. There are some exceptions as GiNaC and Giac, but their design is too complicated for the usual users.

There is not one simple, understandable and modifiable symbolic open source software that is able to compete with the commercial projects in respect of performance. The plan of this project was to fill this blank space.

Sympy–cpp, the symbolic manipulation library, fulfils all original intentions. The simplicity of design and useful functionality with acceptable time of computation was achieved. The source codes of the library have 7804 lines[1] and some more lines are situated in the test framework and benchmarks. It is still a small library. In comparison with software that is developed by more people and also shows longer time of development, it was proved that sympy–cpp is a hopeful project.

There are more things that need to be done, but the library is operational at the present time and state. It is possible to say that sympy–cpp exceeds all expectations.

[1] Comments, empty lines and lines that contains only a right brace are not counted into the sum.

Appendix A

Attached CD

bachelor_thesis.pdf

sympy-cpp.tar.gz

gmp/

- gmp-4.3.1.tar.gz
- gmp-man-4.2.4.pdf

sympy-cpp/

- Add.cpp, Add.h, CompararIx.cpp, CompararIx.h, core.cpp, dep.list, Doxyfile, Ex.cpp, Ex.h, Expr.cpp, Expr.h, functions.cpp, functions.h, Fx.cpp, Fx.h, license, makefile, makefile.old.txt, Mul.cpp, Mul.h, Number.cpp, Number.h, operators.cpp, operators.h, Pow.cpp, Pow.h, Substitution.cpp, Substitution.h, Sym.cpp, Sym.h, tests.h, Utilities.cpp, Utilities.h
- sympy-cpp-lib/
- tests/
  - lines.sh, tests.h, Tdifferentiation.cpp, Tpower.cpp, Tsubstitute.cpp
- documentation/
  - index.html, ...
- bench/
  - Expansion/
    - averages.r, makefile, output_averages_time, test1.ginac.cpp, test1.mathematica.bench, test1.sage.bench, test1.sympy.bench, test1.sympycpp.cpp, est2.ginac.cpp, test2.mathematica.bench, test2.sage.bench, test2.sympy.bench, test2.sympycpp.cpp, test3.ginac.cpp, test3.mathematica.bench, test3.sage.bench, test3.sympy.bench, test3.sympycpp.cpp, test4.ginac.cpp, test4.mathematica.bench, test4.sage.bench, test4.sympy.bench, test4.sympycpp.cpp, test5.ginac.cpp,

test5.mathematica.bench, test5.sage.bench,
test5.sympy.bench, test5.sympycpp.cpp, test6.ginac.cpp,
test6.mathematica.bench, test6.sage.bench,
test6.sympy.bench, test6.sympycpp.cpp, test7.ginac.cpp,
test7.mathematica.bench, test7.sage.bench,
test7.sympy.bench, test7.sympycpp.cpp, test8.ginac.cpp,
test8.mathematica.bench, test8.sage.bench,
test8.sympy.bench, test8.sympycpp.cpp, times

Taylor/

averages.r, makefile, output_averages_time, test1.ginac.cpp,
test1.mathematica.bench, test1.sage.bench,
test1.sympy.bench, test1.sympycpp.cpp, test2.ginac.cpp,
test2.mathematica.bench, test2.sage.bench,
test2.sympy.bench, test2.sympycpp.cpp, test3.ginac.cpp,
test3.mathematica.bench, test3.sage.bench,
test3.sympy.bench, test3.sympycpp.cpp, test4.ginac.cpp,
test4.mathematica.bench, test4.sage.bench,
test4.sympy.bench, test4.sympycpp.cpp, test5.ginac.cpp,
test5.mathematica.bench, test5.sage.bench,
test5.sympy.bench, test5.sympycpp.cpp, times

Appendix B

Used mathematics

Theorem B.1 *Binomial theorem :*
Let $k \in \mathbb{N},\ a, b \in \mathbb{R}$. *Then* $(a+b)^k = \sum_{i=0}^{k} \binom{k}{i} a^i b^{k-i}$, *where* $\binom{k}{i} = \frac{k!}{(k-i)!i!}$.

Theorem B.2 *Multinomial theorem :*
Let $n \geq 2,\ k \in \mathbb{N},\ a_1, a_2, \ldots, a_n \in \mathbb{R}$. *Then* $(a_1 + a_2 + \ldots + a_n)^k =$
$= \sum_{\forall k_1,\ldots,k_n : k_1+\ldots+k_n=k} \binom{k}{k_1,\ldots,k_n} {a_1}^{k_1} \ldots {a_n}^{k_n}$, *where* $\binom{k}{k_1,\ldots,k_n} = \frac{k!}{k_1!\ldots k_n!}$.

Definition B.3 *Taylor series :*
The Taylor series of a real or complex function $f(x)$ *that is infinitely differentiable in a neighbourhood of a real or complex number* a, *is the power series* $\sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k$.

Bibliography

[1] Sympy: *http://code.google.com/p/sympy/*

[2] Wolfarm Mathematica: *http://www.wolfram.com/*

[3] GiNaC: *http://www.ginac.de/*

[4] Sage: *http://www.sagemath.org/*

[5] GNU MP – Multiple Precision Arithmetic Library: *http://gmplib.org*

[6] GMP team: *GNU MP manual*, 2008.