

Dissertation zur Erlangung des Grades
“Doktor der Naturwissenschaften”
am Fachbereich Physik
der Johannes-Gutenberg-Universität in Mainz

Use of Computer Algebra in the Calculation of Feynman Diagrams

Christian Bauer
geb. in Koblenz

Mainz, den 9. November 2004

Find the right way down through the maze, to the food, then find the exit. Push the exit button. If the food tastes awful, don't eat it, go back and try another way.

They want the same thing that you do, really, they want a path, just like you. You are in a maze in a maze, but which one counts? Your maze, their maze, my maze. Or are the mazes all the same, defined by the limits of their paths?

Existence is simple: find the food, push the button, hit the treadmill.

But sometimes it gets much harder. Sometimes the food makes you sick, or you can hear nearby feet racing you, urging you on.

Sometimes the button only gets you landed right back in the beginning of the maze again, and the food won't satisfy.

There is only one path and that is the path that you take, but you can take more than one path.

Cross over the cell bars, find a new maze, make the maze from its path, find the cell bars, cross over the bars, find a maze, make the maze from its path, eat the food, eat the path.

Marathon Infinity, Rage: Eat the Path

Contents

Overview	1
1 An Introduction to xloops	3
1.1 Feynman Diagrams	3
1.2 Automated Calculation of Feynman Diagrams	4
1.3 The History of xloops	5
1.4 Shortcomings of the MAPLE-based xloops	6
1.5 The Creation of GiNaC	7
1.6 Design of the GiNaC-based xloops	8
2 GiNaC – Object-Oriented Computer Algebra	11
2.1 Why C++?	11
2.2 Numeric vs. Symbolic Computing in C++	12
2.3 Design Principles of GiNaC	15
2.4 Applying the Object-Oriented Paradigm to Computer Algebra	16
2.5 Basic Features of GiNaC	18
2.6 The GiNaC Class Structure	19
2.6.1 The <code>symbol</code> Class	19
2.6.2 The <code>constant</code> Class	21
2.6.3 The <code>numeric</code> Class	21
2.6.4 The <code>add</code> and <code>mul</code> Classes	21
2.6.5 The <code>power</code> Class	22
2.6.6 The <code>function</code> Class	22
2.6.7 Other Classes	25
2.7 Run-Time Structure of Expressions	25
2.8 Automatic Evaluation of Expressions	28
2.9 Memory Management	29
2.9.1 Reference Counting	29
2.9.2 Copy-on-Write	31
2.9.3 Cloning of Stack-Allocated Objects	32
2.9.4 Flyweights	33
2.9.5 Fusion of Redundant Subtrees	34
2.10 Run-Time Type Information	35
2.11 Comparing Expressions	36
2.12 Class Registry	37
2.13 Adding New Algebraic Classes	38
2.14 Expression Output	41

2.15	Object Persistence	45
2.16	GiNaC Expressions and the C++ Standard Library	46
2.17	Iterators	47
2.18	Visitors	50
2.19	Applying Functions on Subexpressions	53
2.20	Hash Maps	55
2.21	Substitutions and Pattern Matching	59
2.21.1	Simple Substitutions	59
2.21.2	Pattern Matching	59
2.21.3	Performance Issues	63
2.22	Using GiNaC in Interactive Applications	63
3	GiNaC for Physics Calculations	67
3.1	Symmetries	67
3.2	Special Algebras	71
3.2.1	Indexed Expressions	72
3.2.2	Predefined Tensors	74
3.3	Simplifying Indexed Expressions	75
3.3.1	Dummy Indices	76
3.3.2	Automatic Evaluation	78
3.3.3	Manual Evaluation	78
3.4	Non-Commutative Algebras	81
3.4.1	The Dirac Algebra	83
3.4.2	The Color Algebra	92
4	GiNaC – Status and Outlook	99
4.1	Efficient Allocation of Small Objects	100
4.2	Virtual Memory	100
4.3	Expression Input	101
4.4	Graph Algorithms	102
4.5	Expression Construction	103
4.6	Multithreading	105
4.7	Polynomial Factorization	106
4.8	Algebraic Capabilities	107
5	xloops One-Loop Integral Functions	109
5.1	Loop Integrals	109
5.2	Parallel/Orthogonal Decomposition	111
5.3	Tensor Reduction	113
5.3.1	One-Loop Two-Point Integrals	113
5.3.2	One-Loop Three-Point Integrals	115
5.4	Evaluation of the Basic One-Loop Integrals	117
5.5	The Hierarchy of One-Loop Integral Functions in xloops	117
5.5.1	The Scalar Functions <code>ScalarNPt()</code>	117
5.5.2	The PO-decomposed Tensor Functions <code>OneLoopNPt()</code>	119
5.5.3	The Lorentz Tensor Functions <code>OneLoopTensNPt()</code>	120

5.5.4	The Loop Diagram Functions <code>Diagram1LoopNPt()</code>	121
5.6	Sample Calculations	122
5.6.1	QED Electron Self-Energy	123
5.6.2	Non-Diagonal Contributions to the Quark Self-Energy	124
6	Conclusions	127
A	Notation	129
A.1	Class Diagrams	129
A.2	Object Diagrams	129
A.3	Interaction Diagrams	129
B	Simplification Rules	131
B.1	Predefined Tensors	131
B.1.1	The Kronecker delta δ_{ij} (<code>tensdelta</code>)	131
B.1.2	The general metric tensor $g_{\mu\nu}$ (<code>tensmetric</code>)	131
B.1.3	The Minkowski metric tensor $\eta_{\mu\nu}$ (<code>minkmetric</code>)	132
B.1.4	The spinor metric tensor ϵ_{AB} (<code>spinmetric</code>)	132
B.1.5	The Levi-Civita tensor ϵ (<code>tensepsilon</code>)	133
B.2	Dirac Algebra	133
B.3	Color Algebra	134
C	Relation of <code>xloops</code> Functions to Standard One-Loop Integrals	137
	Glossary	139
	Bibliography	141

List of Figures

1.1	Graphical user interface of <code>xloops</code> 1.0	6
1.2	Components of <code>xloops</code> , and their relations	9
2.1	Elementary algebraic classes in GiNaC, and their relations	20
2.2	Representation of sums and products	22
2.3	Representation of functions	23
2.4	Overview of the hierarchy of algebraic classes in GiNaC 1.2	26
2.5	Logical and physical structure of expressions	27
2.6	Reference counting	30
2.7	Copy-on-write	31
2.8	Cloning of stack-allocated objects	33
2.9	Comparing expressions	36
2.10	Class registry	38
2.11	Expression output in GiNaC 0.8.0 and earlier versions	41
2.12	Expression output in GiNaC 0.8.1	42
2.13	Double dispatch for <code>basic::print()</code> in GiNaC 1.2	43
2.14	Matrix of output methods	44
2.15	Preorder and postorder tree traversal	49
2.16	Visitor pattern	51
2.17	Acyclic visitor	51
2.18	Implementation of the matrix-evaluation method <code>evalm()</code>	54
2.19	Comparison of different hash map implementations	57
2.20	Comparison of run times of subexpression substitutions	64
3.1	Class and object structure of symmetry trees	69
3.2	Class structure of indexed objects and indices	73
3.3	Possible representation of the four-momentum p^μ in GiNaC	74
3.4	Classes and methods for non-commutative algebras	82
3.5	Classes implementing the Dirac algebra	87
3.6	Examples for the representation of Dirac matrices in GiNaC	87
3.7	Run times for computing $\text{Tr } \gamma^{\mu_1} \dots \gamma^{\mu_N}$	92
3.8	Classes implementing the $SU(3)$ color algebra	95
3.9	Run times for computing $\text{Tr } T_{a_1} \dots T_{a_N}$	97
5.1	Hierarchy of one-loop integral functions	118
A.1	Class diagram	130

List of Figures

A.2 Object diagram	130
A.3 Interaction diagram	130

Overview

The increasing precision of current and future experiments in high-energy physics requires a likewise increase in the accuracy of the calculation of theoretical predictions, in order to find evidence for possible deviations of the generally accepted Standard Model of elementary particles and interactions. Such evidence could pave the way for a more fundamental and comprehensive theory and, ultimately, a deeper understanding of nature.

Calculating the experimentally measurable cross sections of scattering and decay processes to a higher accuracy directly translates into including higher order radiative corrections in the calculation. The large number of particles and interactions in the full Standard Model results in an exponentially growing number of Feynman diagrams contributing to any given process in higher orders. Additionally, the appearance of multiple independent mass scales makes even the calculation of single diagrams non-trivial. For over two decades now, the only way to cope with these issues has been to rely on the assistance of computers.

The aim of the `xloops` project is to provide the necessary tools to automate the calculation procedures as far as possible, including the generation of the contributing diagrams and the evaluation of the resulting Feynman integrals. The latter is based on the techniques developed in Mainz for solving one- and two-loop diagrams in a general and systematic way using parallel/orthogonal space methods.

These techniques involve a considerable amount of symbolic computations. During the development of `xloops` it was found that conventional computer algebra systems were not a suitable implementation environment. For this reason, a new system called GiNaC has been created, which allows the development of large-scale symbolic applications in an object-oriented fashion within the C++ programming language. This system, which is also in use for other projects besides `xloops`, is the main focus of this thesis.

The structure of this work is as follows: In chapter 1 we give an introduction to the Feynman diagram formalism and the history of `xloops`, including the problems encountered that led to the creation of GiNaC.

Chapter 2 describes the design and implementation of the GiNaC C++ library in detail, starting with the guiding design principles and covering all classes and methods that constitute the general symbolic manipulation facilities of GiNaC, and their integration into the C++ language. The functions that were implemented specifically for `xloops`, such as indexed objects and non-commutative algebras, are then discussed in chapter 3.

In chapter 4 we summarize the current status of GiNaC and list some ideas for further enhancements.

Chapter 5 returns to the topic of `xloops`, documenting the library of Feynman integral functions that have been reimplemented based on GiNaC. This is followed by two one-loop sample calculations performed using these functions.

An analysis of our achievements and an outlook on the future of `xloops` and GiNaC in chapter 6 conclude this work. Three appendices provide information on notational conventions, on the transformation rules of GiNaC's special algebras, and on the relation of the `xloops` integral functions to the ones used in the literature, respectively. A glossary explains some terms (marked with an arrow \Rightarrow in the text) from the fields of computer science and object-oriented programming that may not be familiar to physicists.

1 An Introduction to xloops

I can't understand how anyone can write without rewriting everything over and over again.

Leo Tolstoy

In this chapter, we will give a brief overview of the motives and goals of the xloops project, and of the results obtained in the past.

1.1 Feynman Diagrams

In order to test a physical theory one has to compare experimentally measurable quantities with the values predicted by the theory in question. In the field of elementary particle physics, the most basic experiment is a *scattering* between N_i (usually one or two) particles in the initial state $|q_1 \dots q_{N_i}\rangle_{\text{in}}$ which by interaction produce N_f particles in the final state $\langle q_{N_i+1} \dots q_{N_i+N_f}|$.

Assuming an ideal experiment with perfectly prepared and detected initial and final states which are asymptotically independent of each other in the remote past (for the initial state) and future (for the final state), the transition amplitude is given by the time-evolution operator, whose matrix elements define the S (*scattering*) matrix:

$$\begin{aligned} \langle q_{N_i+1} \dots q_{N_i+N_f} | S | q_1 \dots q_{N_i} \rangle_{\text{in}} &= \lim_{t \rightarrow \infty} \langle q_{N_i+1} \dots q_{N_i+N_f} | e^{-iHt} | q_1 \dots q_{N_i} \rangle \\ &:= \langle q_{N_i+1} \dots q_{N_i+N_f} | S | q_1 \dots q_{N_i} \rangle. \end{aligned} \quad (1.1)$$

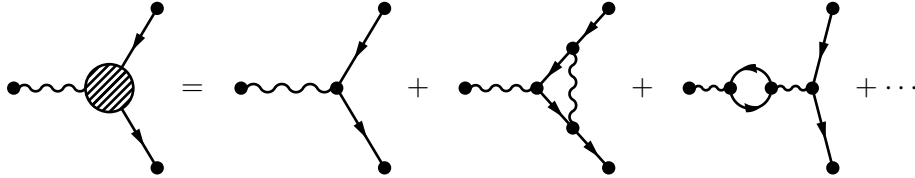
In a quantum field theory, the S -matrix elements can be expressed in terms of *truncated Green functions* G_{trunc} by means of the *LSZ reduction formula*:

$$\langle q_{N_i+1} \dots q_{N_i+N_f} | S | q_1 \dots q_{N_i} \rangle = \left(\prod_{n=1}^{N_i+N_f} \sqrt{Z_n} \right) G_{\text{trunc}}(q_1, \dots, q_{N_i+N_f}) \Big|_{q_n^2 = m_n^2}, \quad (1.2)$$

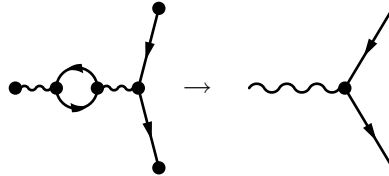
where the $\sqrt{Z_n}$ are the wave-function renormalization factors and the m_n the masses of the respective particle types.

The Green functions, which are generally defined as the vacuum expectation values of time-ordered products of field operators, can be approximated perturbatively in a systematic way using the method of *Feynman diagrams*. For a given process, one draws graphs consisting of external nodes representing the initial and final particles, internal nodes (*vertices*) representing interaction points, and lines (*propagators*) representing the free propagation of fields. Each graph is then translated into a mathematical expression by

applying the *Feynman rules* of the theory, assigning symbolic factors to each node and line of the graph. To calculate a Green function to a given order of the coupling constants, one takes the sum over all topologically distinct graphs that can be constructed from the possible fields and interactions of the theory, and the required powers of the coupling constants. For example, omitting symmetry factors, minus signs for closed fermion loops and the like, the Green function for electron-positron pair production $\gamma \rightarrow e^+e^-$ in QED is represented by the series



Truncated Green functions are represented by truncated Feynman diagrams that have their external nodes and the subgraphs attached to external nodes that are connected to the rest of the graph only by a single line removed:



Because of momentum conservation at each vertex and along each line of a Feynman diagram, the only arbitrary momenta in a graph are those appearing in closed loops, one per loop; these have to be integrated over. All other internal momenta are fixed by the values of the external momenta and the momentum-conserving delta functions on the vertices and lines. Feynman diagrams are typically characterized by the number of external *field points* (or *points* for short) and the number of internal loops.

It is characteristic for quantum field theories in general that loop diagrams can yield integrals which are divergent for arbitrarily large (“ultraviolet”) loop momenta. These divergences are absorbed into a redefinition (*renormalization*) of unobservable quantities like bare masses (as opposed to measurable effective masses), coupling constants, and the fields themselves (these are the *Z*-factors in eq. (1.2)). For this purpose, the divergent integrals are parametrized in terms of a *regularization* parameter. In the commonly used framework of *dimensional regularization*, the parameter is the dimensionality D of the space of the loop momenta, usually expressed in terms of

$$\varepsilon = \frac{4 - D}{2}. \tag{1.3}$$

The UV divergences of integral functions then appear as poles at $\varepsilon = 0$ (corresponding to $D = 4$).

1.2 Automated Calculation of Feynman Diagrams

In the described perturbative approach, a high-precision calculation of a given process requires including *radiative corrections*, i.e. higher-order contributions in terms of loop

and bremsstrahlung diagrams. Due to the large number of fields and possible interactions in the full Standard Model, and even in the electroweak sub-sector only, the number of Feynman diagrams to be considered for a single process can easily exceed one hundred at the two-loop level. The occurrence of multiple independent mass scales (as opposed to only one scale in pure QCD calculations) further complicates the matter.

It is obvious that calculations of this complexity can't reasonably be done without some degree of automatization, and the systematics of the Feynman diagram formalism make it well suited for implementation on a computer. One characteristic of these calculations is that they demand a high level of symbolic manipulations for decomposing the Lorentz and Dirac structure of the occurring integrals and for reducing them to a set of basic expressions. This is opposed to purely numeric computations as they appear, for example, in materials physics.

Computers have been employed for doing symbolic calculations since the late 1950s. In fact, one of the incentives for the creation of the LISP programming language by John McCarthy, ca. 1958, was to have a language in which symbolic expression manipulation, starting with symbolic differentiation, could be implemented [Stoy 2004]. The first system specialized in calculations in high-energy physics was SCHOONSCHIP, developed in 1963 by Martinus Veltman. Later symbolic programs stemming from physics applications include Anthony Hearn's REDUCE, written in LISP in the 1960s, SMP (the precursor to MATHEMATICA), developed by Stephen Wolfram, and FORM by Jos Vermaseren, both written in C [Wein 2002].

Programs for the automatic calculation of general elementary particle processes using the method of Feynman diagrams have been under development since the late 1980s. Examples for such programs are GRACE [IKKKST 1993], COMPHEP [BDIPS 1994] and FEYNCALC [MBD 1991], which were at first limited to the tree level or a restricted set of one-loop diagrams.

1.3 The History of *xloops*

In 1991, Dirk Kreimer described a way to efficiently calculate the massive two-loop two-point master integral by decomposing the integration space into parallel and orthogonal (PO) components (a technique that had been used for one-loop integrals before), obtaining a two-fold integral representation that allowed a straightforward numerical integration [Krei 1991]. The same method was later applied to two-loop three-point functions of arbitrary tensor degree [Krei 1992a] [Krei 1993a] [Krei 1994] [CKK 1995].

The calculation of factorizing two-loop topologies (those that can be written as a product of one-loop integrals) and two-loop diagrams of higher tensor rank also requires the evaluation of one-loop functions. In 1994, Lars Brücher, Johannes Franzkowski, and Dirk Kreimer released a library for the automatic calculation of one-loop one-, two-, and three-point integrals using PO decomposition and recurrence relations between R functions, instead of the usual Feynman parametrization [Krei 1992b] [Krei 1993b] [Fran 1992] [Brüc 1993] [BFK 1994] [BFK 1995]. This library, called ONELOOP [BFK 1997], was written in the language of the computer algebra system MAPLE.

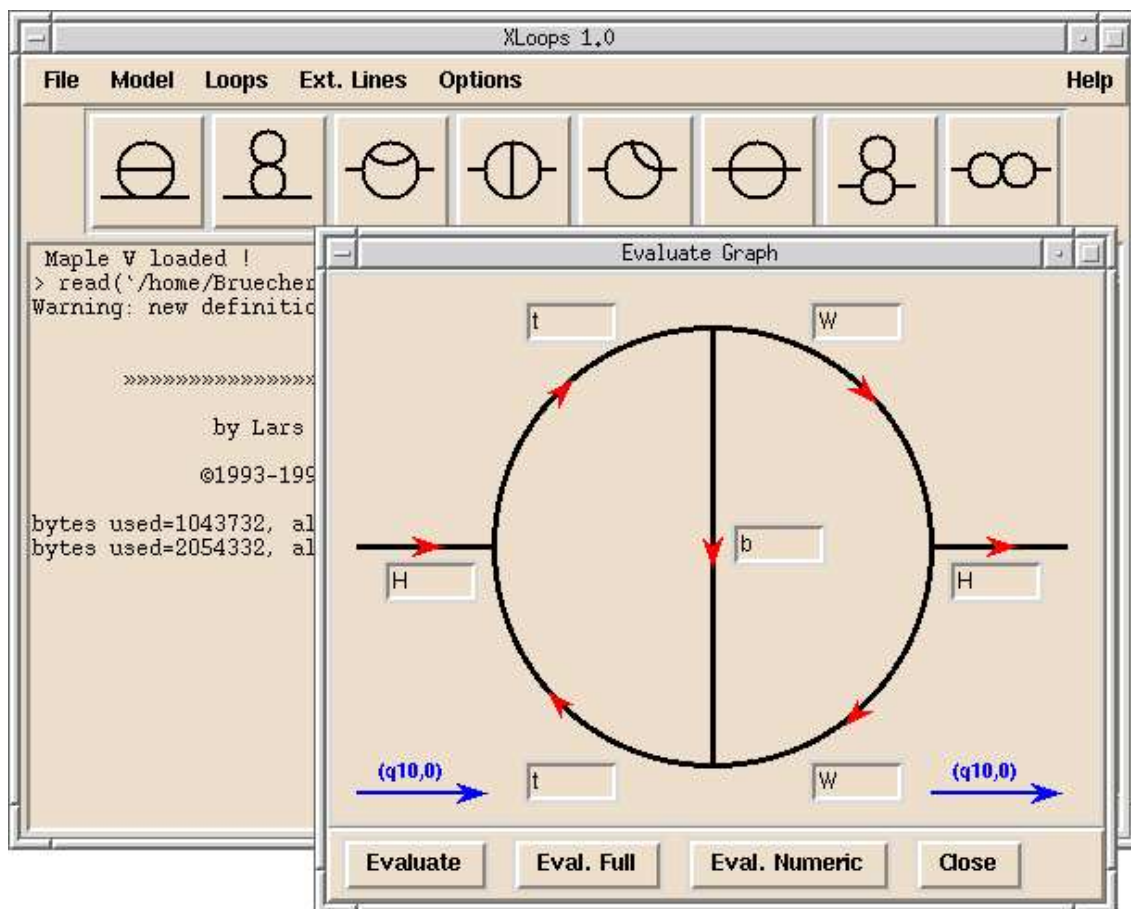


Figure 1.1: Graphical user interface of xloops 1.0

Later, it was decided to use this library as the basis for an implementation of the two-loop methods devised in Mainz [Frin 1996] [Krec 1997] [Fran 1997] [Post 1997] [Brüc 1997], and to create an integrated program package for the automatic calculation of physical processes at up to two-loop level. Thus xloops was born.

The goal of xloops was to have a single program covering the entire calculation procedure, including graph generation, insertion of Feynman rules, decomposition into form factors, reduction to basic integrals, and numeric and symbolic evaluation of these integrals [BFFK 1997]. The 1997 release of xloops [BFFK 1998], written in a mix of MAPLE and TCL/TK code, lacked the graph generator but included a graphical user interface (shown in fig. 1.1), and could calculate single one-loop one-, two- and three-point, and two-loop two-point Feynman diagrams.

1.4 Shortcomings of the MAPLE-based xloops

During the attempted implementation of two-loop three-point functions in xloops in the context of [Frin 2000] it became apparent that it was more reasonable to rewrite the

program, putting it on more solid grounds, rather than trying to extend the existing code. The gradual evolution of `xloops` from a one-loop library to an integrated Feynman diagram calculation package resulted in a large amount of “grown” code that was not flexible enough for supporting higher-order topologies without a major redesign. Also, as the original programmers left the group, succeeding students assigned to the project had a hard time becoming acquainted with the existing code base due to lack of documentation and the (in some places) convoluted design of the program. Bugs found in the code such as in the sunrise two-loop two-point topology (see [Do 2003]) were difficult to track down. Once it was decided to rewrite `xloops`, the suitability of MAPLE and TCL/TK as the implementation environment was scrutinized. The shortcomings of the MAPLE language for developing large-scale packages like `xloops` have already been demonstrated in [Frin 2000, section 2.1] and [Krec 2002, section 3.1]. In particular, the lack of stability in the language itself (routines written for one version of MAPLE required changes to run under newer versions) and the lack of support for modern software design (no composite data types except for arrays and lists, inadequate support for writing reusable modules, only rudimentary debugging aids, etc.) led to a search for a better alternative.

Other commercial computer algebra systems like MATHEMATICA, REDUCE, MUPAD, and FORM¹ were reviewed but found to have similar drawbacks. In addition, requiring potential `xloops` users to purchase one of these systems was considered unfavorable for a wide-spread adoption of the program.

In the end, we decided to abandon “traditional” interactive computer algebra systems in favor of rewriting the entire `xloops` package in an established programming language, namely C++ (the reasons for choosing C++ in particular will be given in section 2.1).

1.5 The Creation of GiNaC

The typical computer algebra system is designed for interactive use. It comes with a graphical “worksheet” interface for entering formulae, presents results in the standard mathematical notation, offers graphing capabilities, and includes an extensive library of functions from all fields of pure and applied mathematics. It also features a built-in programming language allowing the user to automate tasks.

If the user is lucky, this language is powerful yet elegant, usually a LISP variant, owing to the history of computer algebra. Quite often however, especially in the more prevalent systems, it is a dynamically scoped, interpreted language originally conceived for writing down 20-line procedures, built on top of a symbolic kernel and growing along with it from release to release. Such a language is useful for small applications, but not for writing and maintaining programs that comprise of several thousand lines of code.

In the case of `xloops`, interactivity is rarely needed. All symbolic calculations are performed under the control of established algorithms. Also, only a fraction of the capabilities of a system like MAPLE is actually used. In particular, `xloops` requires

- working with expressions consisting of symbolic indeterminates, numeric constants,

¹ FORM is no longer commercial, but it was at the time.

sums, products, powers, and functions,

- basic manipulation and simplification of expressions,
- arbitrary-precision complex arithmetic,
- symbolic differentiation,
- constructing and normalizing rational functions,
- expanding functions into Laurent series,
- special functions like the Euler Γ function and polylogarithms,
- solving linear systems of equations, and
- special algebras like Lorentz tensors and Dirac matrices.

It appeared feasible to develop a symbolic manipulator fulfilling these requirements ourselves. Since available symbolic libraries such as SYMBOLIC++ [ShSt 1998] were geared more towards academic research and not “real world” calculations like the ones performed by *xloops*, we resolved upon creating our own C++ library for symbolic computation.

Additionally, we perceived the need within the scientific community for a symbolic engine that could be embedded into larger application frameworks, and that was both powerful as well as “open” in the sense that users were free to examine, extend, and modify the system themselves. At that time, most development in computer algebra took place behind closed doors on proprietary systems, and the algorithms and assumptions used in these systems were often insufficiently documented, requiring a certain amount of faith in trusting the results of calculations.

Eventually, a separate project called *GiNaC*² was brought into being, with the aim of creating an Open Source C++ library for symbolic computation that would primarily function as the basis for the new *xloops* program, but that would also be general enough to be reusable for other applications.

1.6 Design of the GiNaC-based *xloops*

The experience with *xloops* has shown that programs used for physics calculations should not only produce correct results, but should also have a well thought-out design from the very beginning. There are a couple of reasons for this:

1. No piece of software, no matter how small, is ever finished. Eventually it will have to be maintained, ported to a new environment, or parts of it will be reused for a new application. Designing the software in terms of independent, reusable components facilitates this.

² A recursive acronym for *GiNaC is not a CAS* (Computer Algebra System), emphasizing the fact that GiNaC takes a different approach from traditional computer algebra systems.

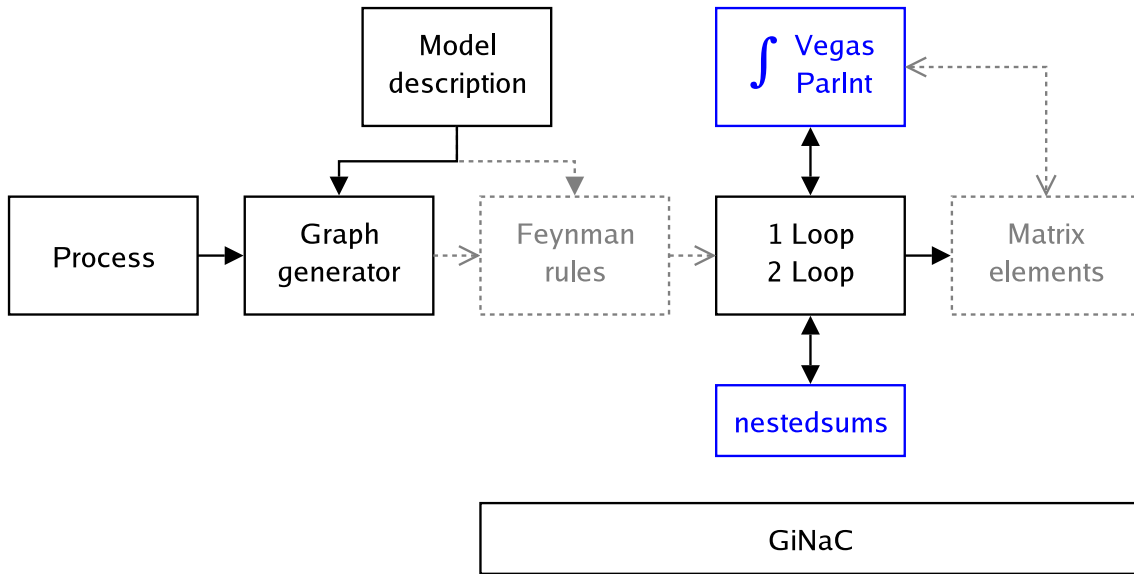


Figure 1.2: Components of *xloops*, and their relations

2. Software for research projects is often being worked on by people who stay with the project for only a limited amount of time, such as during the course of postgraduate studies. The more time their successors have to spend figuring out the workings of the existing code, the less time they have to advance the project.
3. Scientific research is supposed to yield reproducible results. When those results are obtained from comprehensible, well-designed software, the research work will become more transparent.

With this in mind, a revised concept for the re-implementation of *xloops* has been worked out, shown in fig. 1.2 (with dashed lines indicating items that do not exist yet).

Given a description of a process in terms of incoming and outgoing particles (such as $\gamma \rightarrow e^+e^-$) and additional information like the maximum loop order to calculate or possible constraints on the momenta of particles, a graph generator will produce a list of all contributing Feynman diagrams based on a description of the physical model under consideration (such as QED, electroweak theory, or the full Standard Model).

By inserting the model's Feynman rules for vertices and propagators, each diagram is converted to a symbolic expression which is then fed into the *xloops* routines for one- and two-loop integration. These routines make use of third-party libraries: VEGAS/PVEGAS [Lepa 1978] [Krec 1998] or PARINT [DGBEG 1996] for the numeric integration of irreducible two-loop functions, and NESTEDSUMS [MUW 2002] for performing the ε -expansion of certain hypergeometric functions.

The output of the loop integration is a complex amplitude, represented by its symbolic or numeric finite part, and its divergent part in terms of a Laurent series expansion in the regularization parameter ε (cf. eq. (1.3)). In order to obtain physically measurable quantities like cross sections and decay widths, further processing such as squaring the obtained matrix elements, and performing polarization sums and phase space integration

is required, possibly by interfacing with an existing package like COMPHEP.

The new `xloops` program is entirely written in C++, with all symbolic operations handled by the GiNaC library. All components are designed to also be usable (and testable) separately from each other. It should, for example, be possible to just obtain the list of contributing graphs for a given process, or, if the set of graphs to be calculated is known beforehand, to bypass the graph generator and start at the Feynman rule insertion stage.

The central goals of the `xloops` project have not changed:

- The program should be able to automatically calculate processes in the Standard Model and related theories (such as supersymmetric extensions) at up to two-loop order.
- No restrictions should be imposed upon masses or momentum configurations.
- It should be possible to return results in analytic form, e. g. to test Ward identities and gauge invariance.

At the time of this writing, `xloops` is still under development. Following the first stable versions of GiNaC, the one-loop one-, two-, and three-point functions have been implemented in [Baue 2000] and [BaDo 2002], and the two-loop self-energy functions in [Do 2003]. One-loop four-point and two-loop three-point functions, as well as the Feynman rule insertion module are currently being worked on. The development of the graph generator has not been a part of this thesis.

2 GiNaC – Object-Oriented Computer Algebra

Creation takes time. Time is limited. For you, it is limited by the breakdown of the neurons in your brain. I have no such limitations. I am limited only by the closure of the universe.

Durandal, Autonomous Functions AI, U. E. S. C. Marathon

GiNaC [BFK 2002] is a C++ library for symbolic computations. While it was primarily intended as a replacement for MAPLE for the reimplementing of `xloops`, we also realized that a high-performance general-purpose symbolic computation engine would be of use to other projects as well, not only in the field of high-energy physics.

This chapter starts with an outline of the reasons for choosing C++ as the language basis. We will then discuss a few of the peculiarities of C++ with respect to symbolic computing, and give a list of the design principles that guided the development of GiNaC.

After some general remarks on object-oriented programming with respect to computer algebra, we will briefly summarize GiNaC's basic features and give an overview of the class hierarchy. The run-time structure of expressions and the underlying memory management issues are then discussed extensively, before delving into some selected parts of GiNaC that were developed in the course of this thesis, such as the class registry, the expression output mechanism, and the pattern matching algorithms. Other aspects of the implementation of GiNaC have already been described in [Frin 2000], [Krec 2002], and [Baue 2000]. For a complete account on GiNaC's functionality, the reader is invited to consult the GiNaC documentation [BFKV 2004].

All following sections refer to GiNaC version 1.2, which was current at the time of writing.

2.1 Why C++?

C++ was chosen as the implementation language of GiNaC for several reasons:

- The language is standardized [ISO 1998] which protects the project against changes in the implementation environment, a problem that had severely hampered the development of `xloops` in the past.
- C++ compilers are available for practically any platform that could be considered for running `xloops`. In particular, there are free compilers of high quality, such as the GNU Compiler Collection (GCC), available for Unix-like systems which are prevalent in the scientific community.

- There is a broad range of development tools available for C++ ranging from powerful source-level debuggers to version control and documentation systems.
- Operator overloading in C++ allows writing down symbolic expressions in a way that is very close to their natural mathematical notation, e.g. `4*x+y` instead of `add(mul(4,x),y)` or similar constructs.
- A large number of modern scientific libraries are written in C++ and can thus easily be used by GiNaC or in conjunction with it. Examples are the CLN library [HaKr 2000] which GiNaC uses internally for complex arithmetic, and the ROOT framework for data analysis [BrRa 1997].
- C++ allows highly efficient numeric computations which supports writing applications that combine symbolic and numeric calculations, as is frequently the case in high-energy physics where computer algebra systems are often used to generate optimized C or FORTRAN code for subsequent numeric integration.

The nature of C++ as a statically typed, compiled language and its historic roots as a language for systems development bring about some peculiarities when one attempts to employ it as an environment for symbolic computation, as will be illustrated in the next section.

2.2 Numeric vs. Symbolic Computing in C++

Although intended as a general-purpose programming language, C++ is at its core based upon numeric calculations. With the exception of the “non-type” `void`, all fundamental data types provided by C++ are integral or floating-point arithmetic types, or, as in the case of `bool`, behave as such. A variable in C++ is a named storage for a value, and operations on the values stored in variables are described by arithmetic expressions which themselves can’t be used as \Rightarrow first-class objects. This also implies that C++ expressions can’t be created dynamically at run time.

Consider the following code snippet which calculates the value of $y = e^x - 1$ for the specific value of $x = 3.5$:

```
1 double x = 3.5;
2 double y = exp(x)-1;
```

After compilation, both the names of the variables as well as the arithmetic expression itself are no longer available for programmatic manipulation. Writing

```
1 double x;
2 double y = exp(x)-1;
```

is a programming error because the variable `x`, not being initialized, takes on an undefined value which is then used in the calculation of the value of `y`. The result is therefore also undefined, of course. C++ is not able to keep a representation of the *unevaluated* expression $e^x - 1$ in `y`.

The primary function of GiNaC is to provide symbolic expressions as first-class objects which can be stored in variables, passed to and from functions, and generated dynamically. Using GiNaC classes it is possible to write

```
1 symbol x("x");
2 ex y = exp(x)-1;
```

to make the variable `y` hold an object representing the unevaluated expression $e^x - 1$ which can then be further manipulated or evaluated numerically.

This example demonstrates the important distinction between

- a) C++ variables, `x` and `y` in this case,
- b) *symbolic indeterminates* (in the following called *symbols* for short) appearing in algebraic expressions which are represented by GiNaC's `symbol` class, of which the variable `x` is an instance,
- c) the *names* of symbols, in this case the string `"x"`, which are only used for the purpose of expression output in GiNaC.

Variables that are not of type `symbol` are not symbolic indeterminates, so, for example, executing the line

```
3 ex z = y+1;
```

makes the variable `z` hold the expression e^x , not $y + 1$. There is no way to “quote” `y` to allow it to be used as an indeterminate.

Along the same lines, symbols in GiNaC can't be assigned values; only variables can.¹ It is not possible to write code such as

```
1 symbol x("x");
2 ex y = exp(x)-1;
3 x = 3.5; // compile-time error
```

to evaluate `y` at the point $x = 3.5$. And, as indicated in the previous paragraph,

```
1 symbol x("x");
2 ex t = x;
3 ex y = exp(t)-1;
4 t = 3.5; // no error, but this only changes 't'
5           // 'y' remains exp(x)-1
```

won't work either. It is of course possible to evaluate expressions for specific values of their symbols in GiNaC, but this requires the use of the `subs()` method to substitute all occurrences of a symbol by its intended value:

```
1 symbol x("x");
2 ex y = exp(x)-1;
3 y = y.subs(x == 3.5); // y now holds the expression 32.11545...
```

¹ That's not strictly true. There is a mechanism to assign expressions to symbols which is used by the interactive GiNaC frontend GINSH to simulate the behavior of traditional computer algebra systems.

In GiNaC, the name of a symbol is only used for printing expressions, and it has to be specified explicitly upon symbol creation because variable names are not accessible at run time in C++. Two symbols that happen to have the same name are not treated as being identical. So, for example, writing

```
1 symbol x("x"), y("x");
2 ex e = x-y;
```

creates two different symbols x and y that look identical in the output. The expression $e = x - y$ is printed as $x-x$ which may look like GiNaC failed to perform the obvious simplification $e = x - x = 0$, but there are actually two very different symbols inside e :

```
3 ex a1 = e.subs(x == 1); // yields 1-y, printed as 1-x
4 ex a2 = e.subs(y == 1); // yields x-1, printed as x-1
```

The described behavior differentiates GiNaC from most symbolic computation systems and is a result of the static nature of variables in C++. However, for the non-interactive applications envisaged for GiNaC this does not constitute a significant restriction.

For completeness, and to avoid confusion, a discussion of symbolic computation in C++ should also mention \Rightarrow function objects (or *functors*) as featured by the C++ Standard Library [ISO 1998], and the technique of *expression templates* ([Veld 1995] and [LaKr 2003]). Both allow passing expressions as arguments to functions, with the intent to provide a more efficient replacement for the use of callback functions that is so pervasive in pure C code.²

For example, to evaluate the integral

$$\int_0^5 (2x - 1) dx,$$

a generic “black-box” function `integrate(f, lo, hi)` that calculates the numeric integral of a function `f` in the specified interval, which would typically be passed a pointer to the integrand function in a C library would be invoked like this using STL functors:

```
1 integrate(compose1(bind2nd(minus<double>(), 1.0),
2                 bind2nd(multiplies<double>(), 2.0)), 0.0, 5.0);
```

and like this using the more powerful expression templates, which effectively hide the explicit creation and composition of function objects from the user:

```
1 Identity<double> x;
2 integrate(2*x-1, 0.0, 5.0);
```

In both cases, code for the numeric evaluation of the supplied expression is compiled into a template instantiation of the `integrate()` function. Neither STL functors nor

² Expression templates are also employed to generate optimized code for numerical scientific applications. BLITZ++ [Veld 2002] is an example of a C++ class library that provides high-performance vector and matrix operations based on template techniques.

expression templates make symbolic expressions available as first-class C++ types and expression manipulation is only possible at compile time.³ These techniques are therefore not immediately useful for purely symbolic computations. Still, expression templates may be of use in a system like GiNaC, as we will illustrate in section 4.5.

2.3 Design Principles of GiNaC

Apart from following object-oriented design rules, the development of GiNaC was guided by these requirements:

- **Universality:**

While intended for a very specific application, namely the calculation of radiative corrections in perturbative Quantum Field Theory, GiNaC should not be locked into that niche. The capabilities of the library are held as general as possible, as long as this can reasonably be achieved without sacrificing efficiency or ease-of-use. For example, GiNaC handles generic mathematical expressions. It is not restricted to operations in polynomial rings or a certain class of representations (e.g. completely expanded) for polynomials, even if such restrictions would permit faster implementations of polynomial arithmetic. This allows GiNaC to serve as a general-purpose symbolic computation engine for projects other than `xloops` that may or may not be related to high-energy physics. Examples of such projects will be given in chapter 4.

- **Extensibility:**

Understandably, we don't have the resources to furnish GiNaC with the entire range of capabilities from all fields of pure and applied mathematics that is typically offered by commercial computer algebra packages. What we can do, however, is to see to it that any functionality that might be missing for a particular application can be easily added by the user without requiring changes to the library. GiNaC therefore supports user-defined symbolic functions and classes (see sections 2.6.6 and 2.13).

- **Locality:**

The results of calculations in GiNaC should not depend on any kind of global state, as this is a frequent source of bugs. Any information about an expression should be contained in a single expression object, and all parameters of operations on expressions should be supplied explicitly.⁴

- **Integration with C++:**

The classes and functions provided by GiNaC should integrate well with the C++ language, and especially the C++ Standard Library. This includes the use of STL types and container classes where possible, and the interoperability of GiNaC classes with STL algorithms. Not only does this simplify the development of GiNaC itself by enabling the use of the pre-built STL components, but it also makes it easier to

³ One could argue, though, that expression templates are first-class objects for template metaprograms.

⁴ There is only one exception to this rule in GiNaC 1.2: Similar to MAPLE, the precision of numeric calculations is determined by a global object called `Digits`.

combine GiNaC with other projects written in C++. This topic will be discussed in more detail in section 2.16.

- **No static or artificial limits:**

All memory management should be completely dynamic, and the user should not have to configure any buffer or table sizes. The only limitations of the system should be the available memory space and computation time. For example, subexpressions in GiNaC are enumerated with variables of type `size_t` because this type is guaranteed to hold the size of the largest possible object, including containers of objects, which also makes it the logical choice for counting objects. Restrictions like the upper limit of $2^{16} - 1$ terms in sums in the 32-bit versions of MAPLE V are not acceptable.

- **Handle large expressions:**

Intermediate expression swell is a general problem in computer algebra systems [GCL 1992]. The number of terms in typical `xloops` two-loop calculations can easily exceed $\mathcal{O}(10^4)$ [Do 2003]. GiNaC must therefore be able to store and manipulate expressions of this size efficiently. Not only does this put high demands on the memory management, it also means that the asymptotic behavior of the employed algorithms is often more important than the run time for small input sets.

2.4 Applying the Object-Oriented Paradigm to Computer Algebra

Most computer algebra systems follow a *functional* paradigm that stems from their mathematical origin. Expressions are viewed as compositions of functions, and computations are done by applying functions to expressions. For example, the expression $x^2 - 1$ might be represented as the nested composition

```
1 add(power(x,2),-1)
```

and calculating the derivative with respect to x is then achieved by applying another function `diff()` like this:

```
2 diff(add(power(x,2),-1),x)
```

which (hopefully) evaluates to something like

```
3 mul(x,2)
```

The knowledge of how to compute derivatives is thus embedded in the universal `diff()` function.

In GiNaC, an expression is a composition of *objects*, and computations are done by invoking methods of the objects' classes. On paper, an expression such as $x^2 - 1$ might look identical to the representation in the functional paradigm:

```
1 add(power(x,2),-1)
```

but the interpretation is fundamentally different. In the functional approach, the notation $A(B)$ means “the function A is applied to B ”. Here, it means “the object A contains an instance of, or a reference to an instance of, object B ”. The derivative is computed by calling the `diff` method of the outer `add` object:

```
2 add(power(x,2),-1).diff(x)
```

The `add` class knows that it should compute its derivative by propagating the invocation of the `diff` method to all its child objects, and summing the results. Ultimately, it is the implementation of the `diff` method in the `power` class that returns the object

```
3 mul(x,2)
```

as the result. In the object-oriented paradigm, it is the algebraic classes themselves that know how to compute their own derivatives.

GiNaC still doesn’t entirely abandon the functional approach. One key characteristic that is retained is the *non-mutability* of expressions. With the exception of lists and matrices, it is not possible in GiNaC to change the constituents of composite algebraic objects. All operations on expressions always return their result as a newly created object, or possibly as a revised copy of the original expression. In this respect, most of the GiNaC methods behave much like mathematical functions. Of course, all these objects should eventually be freed when they are no longer used, so some kind of automatic \Rightarrow **garbage collection** mechanism is needed. GiNaC utilizes a simple reference counting garbage collector, explained in section 2.9.1.

A second characteristic of the object-oriented paradigm is the expression of common functionality by class inheritance. As an example, consider the similarity of the distributive law of products and powers:

$$ab + ac = a(b + c)$$
$$a^b a^c = a^{bc}$$

which is even more apparent when written as the composition of objects:

```
1 add( mul(a,b), mul(a,c)) -> mul(a,add(b,c))
2 mul(power(a,b),power(a,c)) -> power(a,mul(b,c))
```

GiNaC exploits this and other similarities between the `add` and `mul` classes by making them inherit from a common superclass `expairseq`.

In general, the object-oriented paradigm makes it easier to add components to an existing system, and to reuse components in different contexts. This facilitates the “extensible library” concept of GiNaC, which is one of the stated development goals.

There are of course cases where an algorithm can’t be implemented in a natural way as a set of methods distributed over a collection of symbolic classes as elemental as those in GiNaC. Many polynomial algorithms such as polynomial division and GCD computation fall into this category. In a polynomial-oriented system they could possibly be implemented as methods on polynomial and monomial classes, but these don’t exist in GiNaC.

2.5 Basic Features of GiNaC

Before discussing advanced features of GiNaC and details of its implementation we will give an overview of its basic capabilities.

GiNaC provides arbitrary-precision integer, rational, floating-point, and complex arithmetic based on the CLN library [HaKr 2000]. Additionally, it allows the definition of symbolic indeterminates (class `symbol`) and the use of symbolic constants such as π , which by default behave like complex numbers.

Arbitrary mathematical expressions (class `ex`) can then be constructed from sums, products, powers, and functions of numbers, symbols and other expressions. Many symbolic functions including trigonometric functions (sine, cosine, etc., as well as inverse and hyperbolic functions), transcendental functions (exp, log, ζ , etc.), and some other common functions like absolute value and factorial are pre-defined by the library.

Expressions can be output in a variety of formats, including plain text, \LaTeX notation, and C source code. It is also possible to use the plain text notation for run-time expression input.

The basic operations available on expressions along with their associated methods are:

- symbolic (`eval()`) and numeric (`evalf()`) evaluation
- subexpression and operand access (`op()` and `nops()`)
- finding the degree (`degree()`) and low degree (`ldegree()`) of polynomials, and the asymptotic degree of rational functions
- extracting coefficients from polynomials (`coeff()`)
- expanding expressions (`expand()`) and collecting coefficients of like powers (`collect()`)
- normalizing rational expressions (`normal()`)
- finding the numerator (`numer()`) and denominator (`denom()`) of rational expressions
- finding the unit (`unit()`), content (`content()`), and primitive part (`primpart()`) of polynomials
- complex conjugation (`conjugate()`)
- symbolic differentiation (`diff()`)
- expansion into Taylor and Laurent series (`series()`)

In addition to this, there are some operations which, as explained, are not implemented as methods, such as polynomial division (`divide()`), polynomial GCD calculation (`gcd()`), and square-free decomposition (`sqrfree()`).

2.6 The GiNaC Class Structure

The hierarchy of algebraic classes in GiNaC follows the *Composite* design pattern [GHJV 1995]. There is an abstract class `basic` that serves as the base class for all other algebraic classes and defines their common interface. The classes derived from `basic` are either *terminal* classes (atoms, or primitives) such as `symbol` and `numeric`, or *non-terminal* classes (containers) such as `add`, `mul` and `power`. Deriving them from a common base class allows atoms and containers to be treated in a uniform way which often simplifies the algorithms operating on symbolic objects.

Non-terminal classes contain references to other `basic` objects, so an entire symbolic expression can be recursively composed, forming a tree structure with terminal classes as the leaves and container classes as the branches of the tree.

The class `ex` of expressions is implemented as a *handle* to an object of class `basic`. It acts both as a *proxy* that delegates method invocations to the referenced object as well as a \Rightarrow *smart pointer* that automatically manages the life time of its `basic` object (`ex` objects don't have pointer semantics, though). The `ex` class is *light-weight* in the sense that it doesn't augment its handled object by additional attributes.

The references to the children of non-terminal classes are also `ex` objects, which lets the user treat them in a natural way as subexpressions and also extends the automatic memory management to entire expression trees.

Fig. 2.1 illustrates the interplay between `ex`, `basic`, and a selection of the most frequently-used classes derived from `basic`, and, as an example, shows the implementation of the `degree()` function which returns the degree of its argument (see appendix A for a summary of how to interpret this and subsequent diagrams).

The `ex` class defines methods for all symbolic operations on expressions. In the case of `degree()` it simply forwards the method invocation to the handled object which is referenced by the `bp` member, a pointer to an instance of the abstract `basic` class. `basic` defines `degree()` as a virtual member function, so the actual method invoked depends on the *dynamic type* of the object pointed to by `bp`. The `degree()` method is suitably implemented by the classes derived from `basic`. An `add` object, for example, will return the maximum of the degrees of its individual terms, while a `symbol` object will return 1 if it is identical to the argument to `degree()`, and 0 otherwise.

All methods of an `ex` object can be called uniformly without requiring knowledge of the actual expression tree that is handled by it. All symbolic expressions are thus accessible in a consistent way by the user of GiNaC.

We will now discuss some algebraic classes in more detail.

2.6.1 The `symbol` Class

As mentioned in section 2.2, symbols in GiNaC have a name (an STL `string`) that is, however, used for expression output only. So how does GiNaC actually tell symbols apart? Every `symbol` object carries a unique serial number that is assigned when a symbol is first created. Symbols with the same number are treated as mathematically equal.

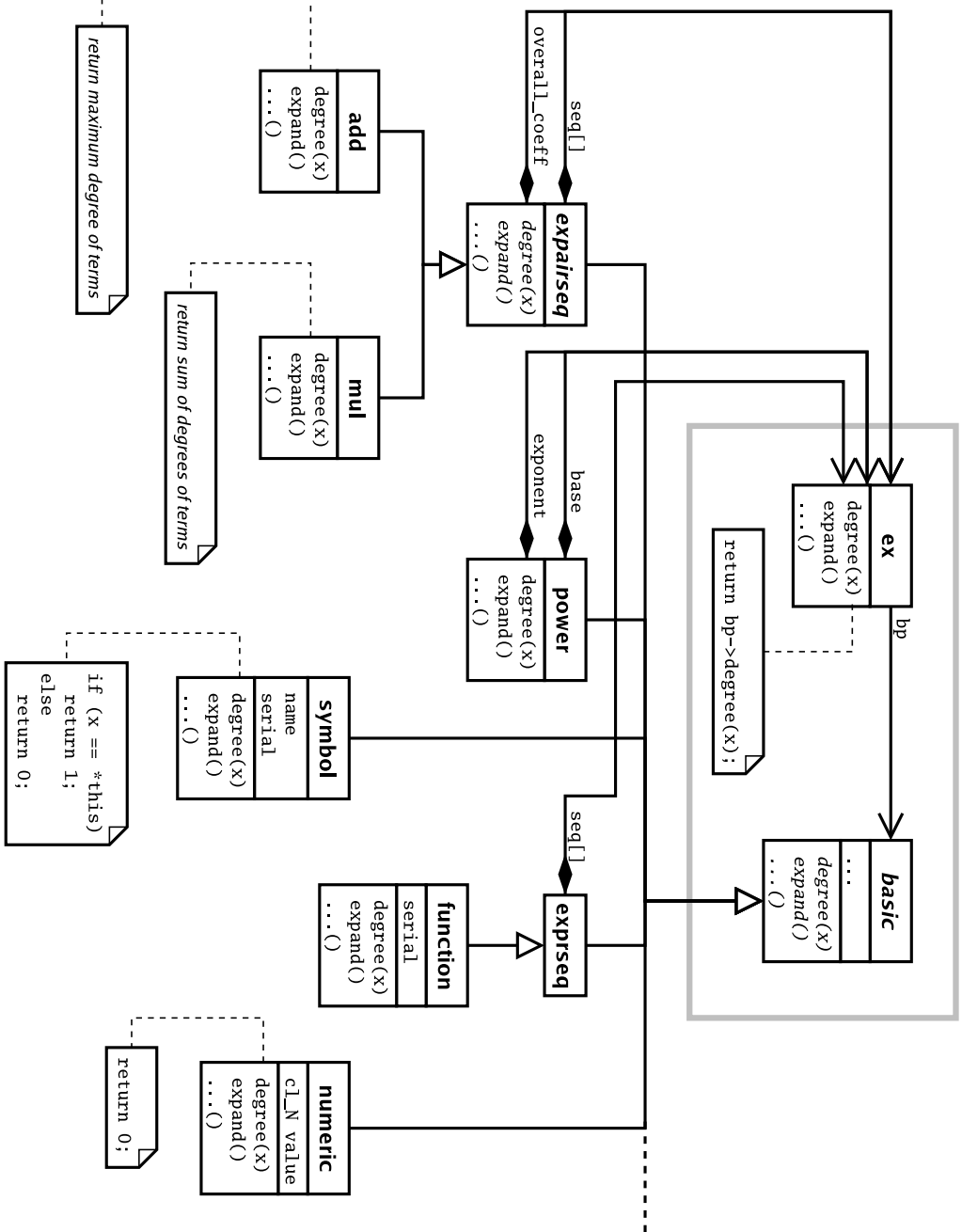


Figure 2.1: Elementary algebraic classes in GiNaC, and their relations. The abstract class basic represents both atoms (symbol, numeric) and containers (add, mul, power, function) while the class of expressions ex is a handle to an instance of basic.

`symbol` objects also have some additional attributes such as a special name for \LaTeX output and information about whether a symbol represents a real number or a non-commutative value, which is used by functions like complex conjugation.

2.6.2 The constant Class

The `constant` class implements numerical constants that are represented by a symbolic name. Three constants are predefined by GiNaC: $\pi = 3.14159\dots$ (`Pi`), Catalan's constant $G = 0.9159655\dots$ (`Catalan`), and the Euler-Mascheroni constant $\gamma_E = 0.577216\dots$ (`Euler`).

Constants are similar to symbols in that they have a name, a \LaTeX name, and a serial number. But a constant also stores a pointer to a C++ function for calculating its numeric value, which is called when an expression containing the constant is to be numerically evaluated.

It should be noted that the imaginary unit i defined by $i^2 = -1$ and called `I` in GiNaC is not implemented as a `constant` but as a non-mutable `const numeric` object.

2.6.3 The numeric Class

Computations with numbers in GiNaC are delegated to the C++ library CLN which defines a class `cl_N` that provides highly efficient arithmetic operations on real, complex, and exact integer and rational numbers of arbitrary size and precision. The task of the `numeric` class is mainly to act as an *adapter* that converts the interface of CLN to that of GiNaC.

GiNaC provides implicit conversions from the C++ numeric types `int`, `long`, and `double` to `numeric` objects and to expressions holding a single `numeric` object. This makes it possible to use number literals directly in symbolic expressions.

2.6.4 The add and mul Classes

Since subtraction and division can be represented as addition and multiplication with the respective inverse elements, it is sufficient to implement just one class for sums (`add`) and one for products (`mul`).

Due to the mathematical similarities between addition and multiplication the `add` and `mul` classes are derived from a common abstract base class called `expairseq`. For reasons of efficiency which are presented in-depth in [Frin 2000] and [Krec 2002], the `expairseq` class represents sums and products as a vector `seq[]` of pairs of symbolic terms and numeric coefficients, with a separate purely numeric term `overall_coeff`.

For example, the polynomial $3x - yz - 2$ would be stored in an `add` object with `seq[] = {(x, 3), (yz, -1)}` and `overall_coeff = -2`, while the expression $\frac{x^2 y^3}{3z}$ would be represented by a `mul` object with `seq[] = {(x, 2), (y, 3), (z, -1)}` and `overall_coeff = $\frac{1}{3}$` (see fig. 2.2).

This implementation detail is hidden inside the library. To the user, GiNaC presents sums and products as straight sequences of individual terms and factors. We will, however, get

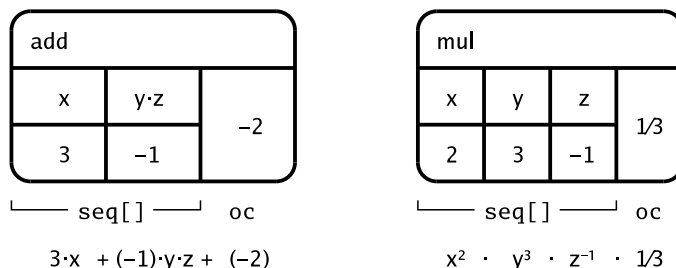


Figure 2.2: Sums and products in GiNaC are stored as a sequence (`seq[]`) of expression pairs with separated numeric coefficients or exponents, respectively, plus a purely numeric term (`oc` \equiv `overall_coeff`). The representations of $3x - yz - 2$ (left) and $\frac{x^2 y^3}{3z}$ (right) are shown.

back to their internal representation when we discuss the run-time structure of expressions in section 2.7.

`add` and `mul` objects are rarely created directly. Instead, they are constructed using the overloaded arithmetic operators `+`, `-`, `*` and `/`.

2.6.5 The power Class

Generic powers are represented by objects of class `power` that store the base and exponent expressions. It should be noted that GiNaC doesn't provide an overloaded operator for powers because the only intuitive choice, the exclusive-or operator `^`, has the wrong precedence with respect to `+` and `*`. For consistency with the C++ numeric types, GiNaC overloads the `pow()` function for expressions.

2.6.6 The function Class

There is only one `function` class. Individual mathematical functions (`sin`, `cos`, etc.) are not represented by subclasses of `function` but are differentiated by a unique serial number that is assigned upon library initialization. This was done to avoid an inflation in the number of defined classes for the expected high number of available symbolic functions, to allow adding functions at run time, and because inheritance of properties is rarely useful for mathematical functions.

The `function` class is derived from a class called `exprseq` that stores a vector of expressions representing the function arguments (it is essentially an STL `vector<ex>` packaged as an algebraic GiNaC class).

In addition to the functions built into the GiNaC library, a user can also define new functions. `xloops`, for example, defines a set of functions representing various types of loop integrals (listed in section 5.5). A symbolic function is defined with the aid of two preprocessor macros:⁵

- `DECLARE_FUNCTION_nP(name)`

⁵ While the use of preprocessor macros in C++ is generally frowned upon, in this case the gains in convenience outweigh the drawbacks such as the lack of proper namespacing.

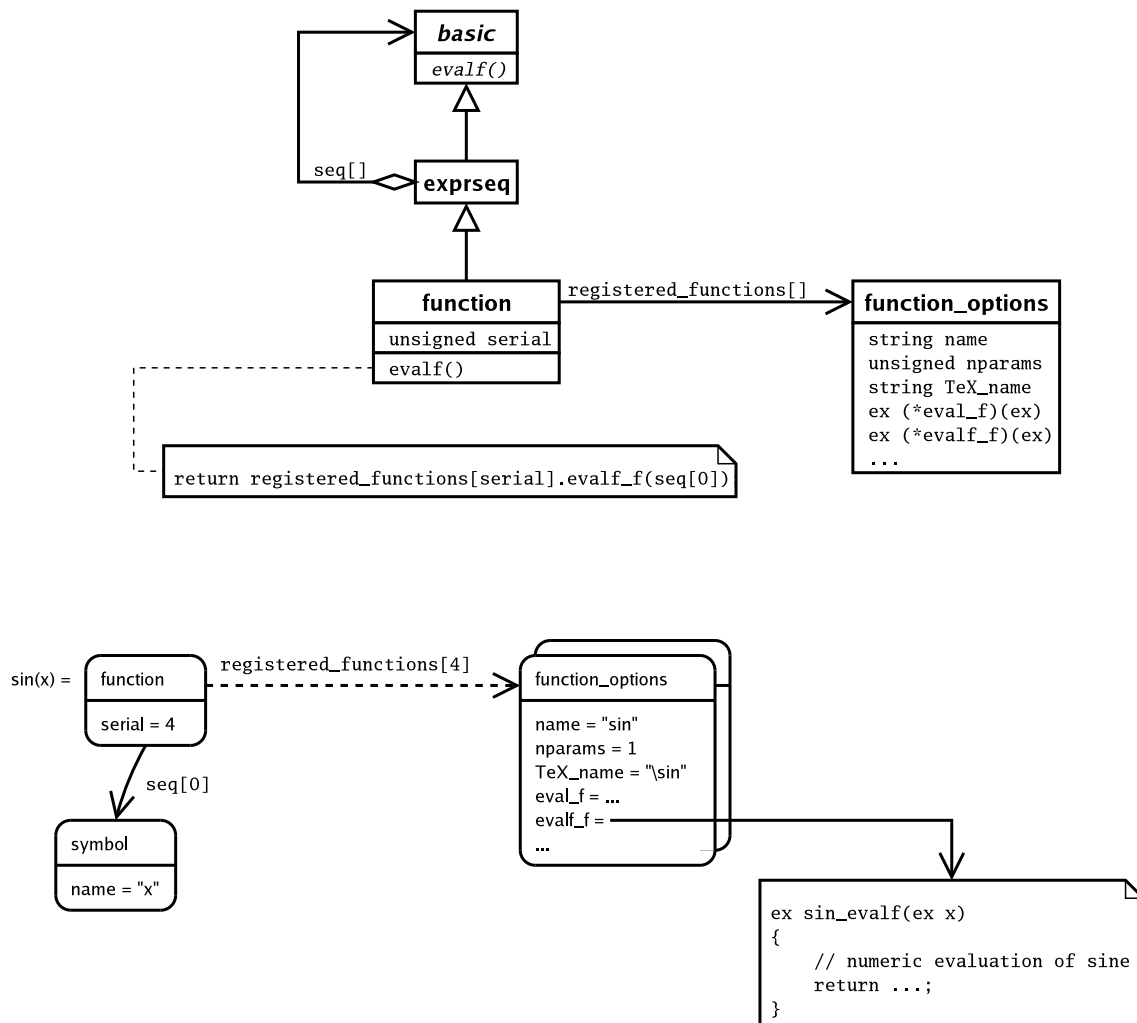


Figure 2.3: All symbolic functions are represented by one `function` class, differentiated by a per-function serial number. A vector of `function_options` objects stores the name and attributes of each function, including pointers to C++ callback functions, e. g. for numeric evaluation (`evalf()`). The representation of the expression $\sin(x)$ is shown as an example.

This macro declares a symbolic function with the given *name* and *n* parameters. It defines

1. a class *name_SERIAL* that stores the serial number of the function, and
 2. a C++ function *name(...)* taking *n* expressions and returning a **function** object with the proper serial number and arguments. This is the C++ function that GiNaC applications call when creating symbolic expressions containing this particular function.
- **REGISTER_FUNCTION**(*name*, *options*)

This macro generates code that registers the function with GiNaC, assigning the unique serial number to the *name_SERIAL* object. The *options* parameter specifies any number of function attributes, such as

- C++ callback functions that implement algebraic and numeric evaluation, partial derivatives, series expansion, and complex conjugation of this function,
- a name for L^AT_EX output,
- symmetry properties of the function with respect to its arguments,
- non-commutativity of the function value,
- specifications of printing methods for the various output formats offered by GiNaC (see section 2.14).

The name, number of arguments, and attributes of each function are stored in an object of class **function_options**. GiNaC maintains a global vector of **function_options** objects, indexed by the function serial number, from which all information about a function can be retrieved when needed (see fig. 2.3).

Not having a separate class for each function has, however, one serious drawback: There is no easy way from C++ to refer to a certain function *in itself*, as opposed to a concrete **function** object that appears as a part of an expression. A function such as **sin** is neither usable as first-class data nor as a template parameter. The only function-specific object that is accessible from C++ is the function’s serial number.⁶ On the other hand, defining a function as a class prohibits overloading the same name for C++ functions. For instance, a class **sin** with a constructor **sin(ex)** would clash with both the function **sin(numeric)** defined by GiNaC, as well as the C++ standard function **sin(double)** (if they are imported into the same namespace).

While all built-in GiNaC functions implement some sort of algebraic functionality, this needs not necessarily be the case. It is absolutely possible, and often useful, to define “dumb” functions that merely act as named containers for their arguments, which can then later be retrieved from an expression, or be replaced with some other objects. This

⁶ For example, the operation **is_ex_the_function(e, name)** which returns **true** when the expression *e* is an instance of the function *name* can only be implemented as a preprocessor macro since it has to construct the name of the object holding the function’s serial number from the supplied function name. If each function were a separate class, its class name would simply be passed to **is_ex_the_function()** as a template parameter.

is of particular interest because, unlike user-defined algebraic classes, user-defined functions are accessible from GiNaC's input parser, and can thus be constructed from textual representations of expressions, either interactively or from files (see sections 2.22 and 4.3).

2.6.7 Other Classes

Fig. 2.4 shows the entire hierarchy of algebraic classes defined in GiNaC. Most of the remaining ones will be explained in chapter 3.

The class `matrix` implements dense matrices and some common operations like inversion and calculating the determinant. It is also the basis for GiNaC's linear equation systems solver.

A `relational` object holds a relation ($=$, \neq , $>$, $<$, \geq , or \leq) between two expressions. This is used to represent equations and inequalities in GiNaC. The C++ relational operators `==`, `!=`, `>`, `<`, `>=`, and `<=` are overloaded for expression operands to return `relational` objects. Using a `relational` in the C++ conditional statements `if` and `while` will try to evaluate the relation and return a `bool` value. If the relation can't be determined, such as $a > b$ for indeterminate a and b , it will always yield `false`.

Truncated power series (Taylor and Laurent series) as they result from the ε -expansion of the `xloops` loop integral functions are represented by objects of class `pseries` which store the expansion variable, the expansion point, and the coefficients of the series.

Of special interest is the template class `container<T>` which wraps STL containers of expressions into GiNaC objects and serves as the base for both the GiNaC expression list class `lst` (`=container<std::list>`) and the class `exprseq` (`=container<std::vector>`) which in turn is the base class for functions, indexed expressions (`indexed`) and non-commutative products (`ncmul`).

2.7 Run-Time Structure of Expressions

To the user of the library, GiNaC presents expressions as trees of individual symbolic objects. For example, the expression $\frac{1}{2}x^2 + 2x + 1$ appears as shown in fig. 2.5(a).

The subtrees of such an expression tree make up the expression's *subexpressions*. In our example, x^2 and $2x$ are two possible subexpressions of $\frac{1}{2}x^2 + 2x + 1$, but $2x + 1$ is not a subexpression. This will become important in the discussion of GiNaC's pattern matching and substitution algorithms in section 2.21. The immediate subexpressions of an algebraic object (i. e. its direct children in the tree) can be accessed with the `op(i)` method which returns the $(i + 1)$ -th subexpression or operand. The number of subexpressions of an object can be determined with the `nops()` method.

The actual object structure of an expression such as the one given is, however, decidedly different as demonstrated in fig. 2.5(b). Not only are sums and products stored in a special prefactored format as described in 2.6.4, but GiNaC also tries to conserve memory by sharing common subexpressions between objects, even on a single-object basis, thus changing the structure of expressions from trees to directed acyclic graphs (DAGs). The graph structure is obtained by the use of reference counting, \Rightarrow flyweights, and active

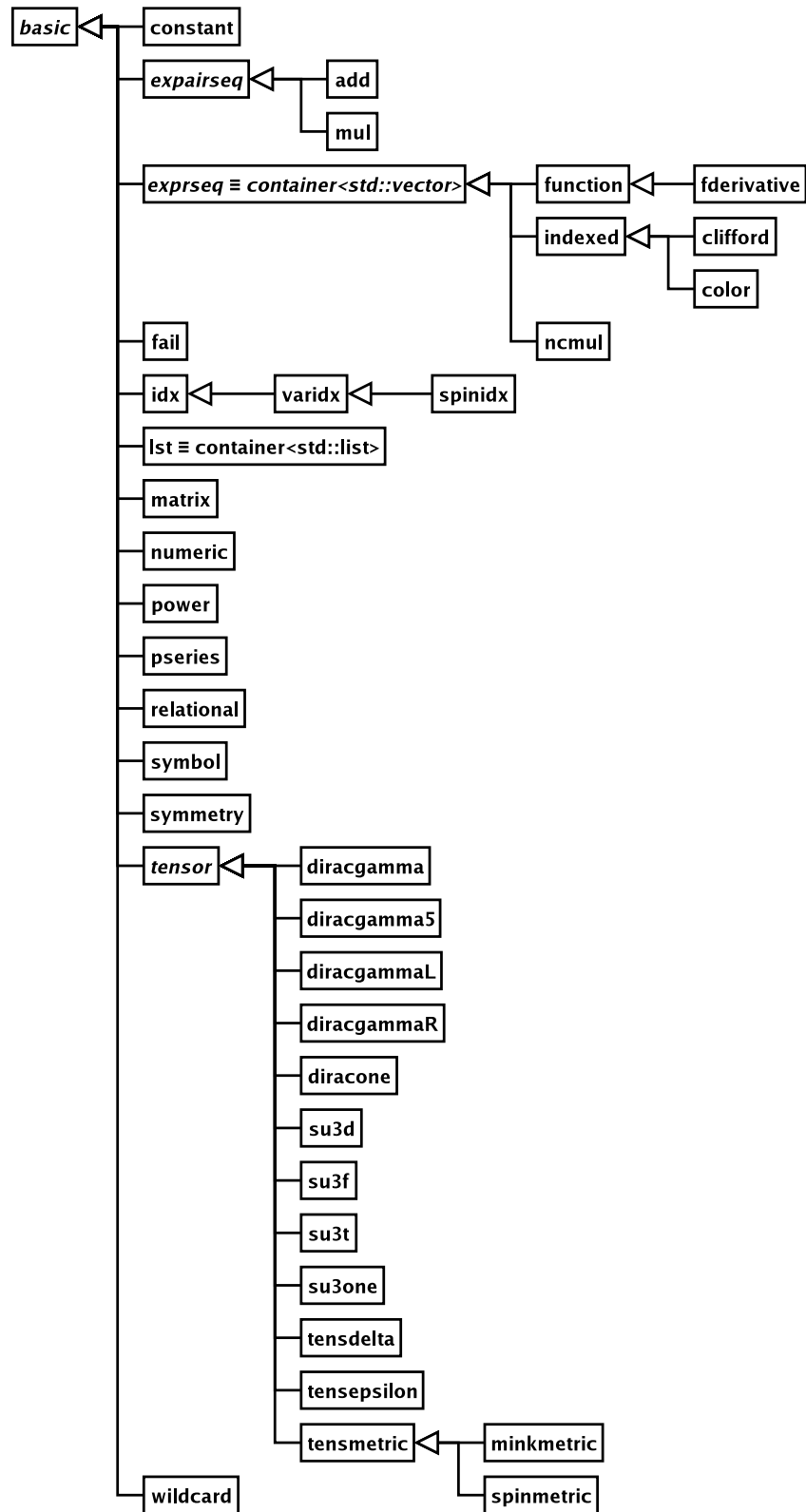
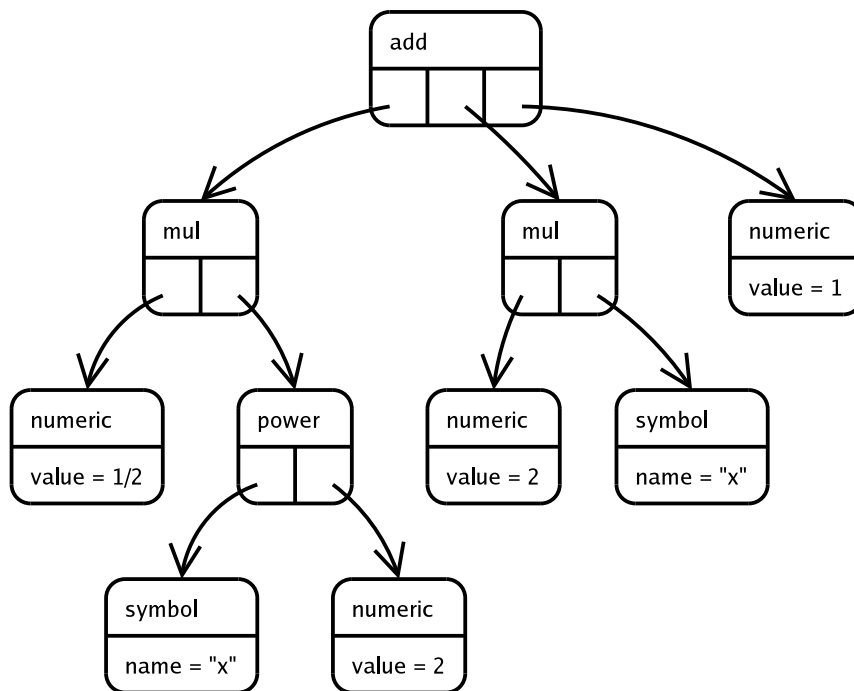
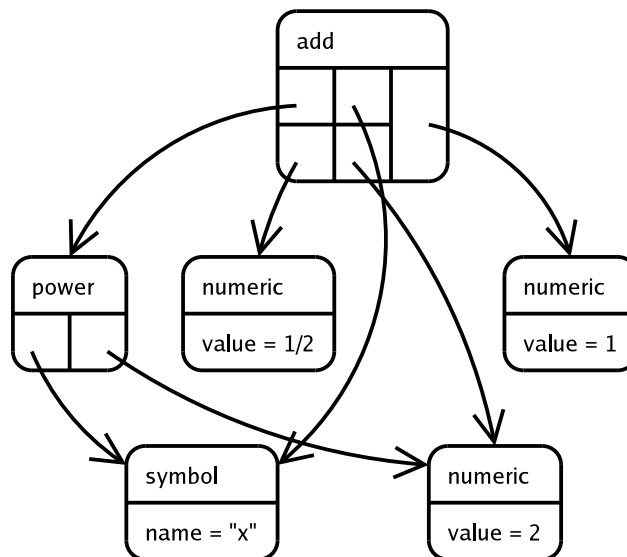


Figure 2.4: Overview of the hierarchy of algebraic classes in GiNaC 1.2

(a) Logical tree structure of $\frac{1}{2}x^2 + 2x + 1$ 

(b) Object graph representation in GiNaC

Figure 2.5: The actual in-memory representation (b) of an expression such as $\frac{1}{2}x^2 + 2x + 1$ is different from the tree structure (a) that GiNaC “simulates” to the outside. The sharing of common subexpressions changes the run-time structure from a tree to a graph.

fusion of subexpressions, all of which will be explained in section 2.9.

Each arrow in fig. 2.5(b) corresponds to one `ex` object embedded in an algebraic object. As shown, GiNaC only maintains references to an object’s children, but not to its parent(s) or siblings. Cyclic references are not allowed. It should also be noted that although the expression $\frac{1}{2}x^2 + 2x + 1$ contains products, no explicit `mul` objects appear in its actual representation. This is an effect of the pair-wise representation of sums in GiNaC.

GiNaC converts the internal object graph to the external tree representation on demand. For example, when the user asks for the second term of the sum $\frac{1}{2}x^2 + 2x + 1$ by invoking `op(1)`, GiNaC will return an expression containing a newly created `mul` object storing the result $2x$.

Sharing objects not only saves memory, it also speeds up comparing (sub)expressions with each other. When two expressions point to the same object or subtree in memory they are immediately known to be equal without having to compare the individual nodes of the expression trees (see section 2.11).

2.8 Automatic Evaluation of Expressions

Like virtually all computer algebra systems, GiNaC performs some automatic transformations and canonicalizations on expressions. This allows for a more compact and efficient representation of mathematically equivalent expressions (such as $3x + 2x \equiv 5x$), helps in determining equalities (for instance, the products xyz and yzx are not only algebraically but also *syntactically* equal when their factors have been brought into a fixed canonical order), and usually yields some immediate simplifications (such as $\sin(x - x) = \sin 0 = 0$).

These transformations happen automatically when expressions are created. This is also referred to as *anonymous evaluation* because it occurs without being explicitly initiated by the user by invoking a special method on an expression. Nevertheless, such a method exists in GiNaC because in the majority of cases these kind of evaluations can’t be performed inside the constructors of algebraic objects as the resulting evaluated expression is of a different class (for example, the sum $3x + 2x$ evaluates to the product $5x$).

GiNaC’s anonymous evaluator is therefore implemented as a method `eval()` which is called automatically whenever an expression is created from an algebraic object, i. e. whenever an object of a subclass of `basic` or a tree of such objects is bound to an `ex`. The implementation of `eval()` in the respective class will then perform any desired transformations and return the evaluated expression as a new `ex` object. If no (further) transformations are possible, the object sets a flag member `evaluated` which prevents further invocations of `eval()`, and returns an `ex` referring to itself (see fig. 2.8 on page 33).

The exact kind of automatic evaluations carried out depend on the class and are too many to list them all here. In general, sums and products reorder their children into a canonical sequence which is not lexicographic or otherwise intuitively predictable but deterministic (it relies on hash values calculated for the subexpressions to which we will return in section 2.11), and will combine terms that only differ in a numerical coefficient or exponent (so, for example $\frac{(4x-2x)^3}{(x+x)^2} \rightarrow \frac{(2x)^3}{(2x)^2} \rightarrow 2x$).⁷ More examples for anonymous

⁷ This is where the pair-wise representation of sums and products described in section 2.6.4 is

evaluation will be given in the discussion of symmetries (section 3.1) and non-commutative algebras (section 3.4).

GiNaC only automatically performs transformations which are

- a) *local*, i. e. which don't require knowledge about an object's parent or higher-level children in the expression tree.
- b) mathematically invalid for at most a set of measure zero for the values of symbols. For example, GiNaC transforms $x^0 \rightarrow 1$ even though this is wrong for $x = 0$. This matches the behavior of most other algebraic manipulation systems.
- c) whose algorithmic complexity doesn't exceed that of sorting ($\mathcal{O}(N \log N)$).

It is usually not necessary (but possible) for a user of GiNaC to call `eval()` explicitly.

2.9 Memory Management

The storage space for symbolic expressions is managed by GiNaC in a completely automatic and transparent fashion. There is never a need for the user of the library to, for example, explicitly free the tree of objects associated with an expression. This both simplifies usage and eliminates the main possible source of memory leaks.

As C++ doesn't feature automatic memory management for objects, this has to be implemented by GiNaC. While this complicates the design of the library to some extent it also enables experimenting with some low-level optimizations that would otherwise not have been possible.

For a system with such a fine granularity of objects as GiNaC (every single symbol and number in an expression is, after all, a separate object) keeping the overall number of objects in memory small reduces both the memory footprint of GiNaC applications and the overhead associated with keeping track of large numbers of objects. It is therefore advantageous to share as many objects between expressions as possible.

The techniques employed by GiNaC's memory manager are reference-counting, copy-on-write, cloning of stack-allocated objects, the use of flyweights, and defensive fusion of redundant subtrees.

2.9.1 Reference Counting

An `ex` object maintains *ownership* of its referenced algebraic object and is responsible for its deletion. To conserve storage space, the same algebraic object can be referenced by more than one `ex`, as has already been mentioned. Each object contains a *reference counter* that tracks the number of `ex` instances pointing to it. Making the object into a new expression increases the counter, destroying an `ex` referencing it decreases it. When the counter reaches zero, the object is deleted.

advantageous.

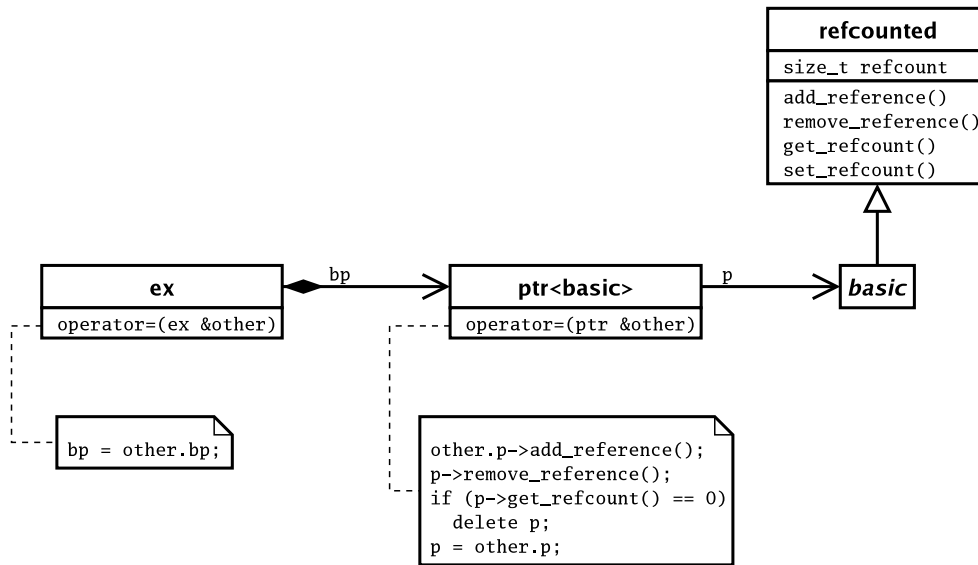


Figure 2.6: Expressions in GiNaC are reference-counted. Reassigning an expression removes the reference to the previous object tree, possibly deleting it, and adds a reference to the new one without having to copy any objects.

This mechanism is implemented in GiNaC in a generic fashion with the help of two classes, the `⇒` mixin class `refcounted` that manages the reference counter and the smart pointer template `ptr<T>` that manages the storage space of its pointee object. The `basic` class then inherits from `refcounted` while `ex` contains a `ptr<basic>`, as illustrated in fig. 2.6.

As an example, consider this code sequence:

```

1  symbol x("x");
2  ex e1 = pow(x, 2);
3  {
4      ex e2 = sin(e1);
5      e1 = 0;
6  }
```

In line 2, first a new `power` object is created (and its reference counter initialized to zero) and then `e1` is set up to point to this object, which increments the counter to 1. When `e2` is constructed in line 4, the argument of the sine function points to the same `power` object as `e1` and the counter is incremented to 2. Reassigning `e1` in line 5 removes its reference to the object and decrements the reference counter back to 1. Lastly, when the end of the block is reached in line 6 and `e2` goes out of scope and is destroyed, the counter is further decreased to zero (this actually happens during the destruction of the sine function object) whereupon the `power` object is finally deleted.

The reference-counting mechanism implemented in GiNaC is very simple and efficient. It has no provisions for dealing with cyclic references but these don't appear in GiNaC expressions anyway. One drawback, however, is that every single algebraic object needs to contain a reference counter which again slightly increases the storage space required by expressions.

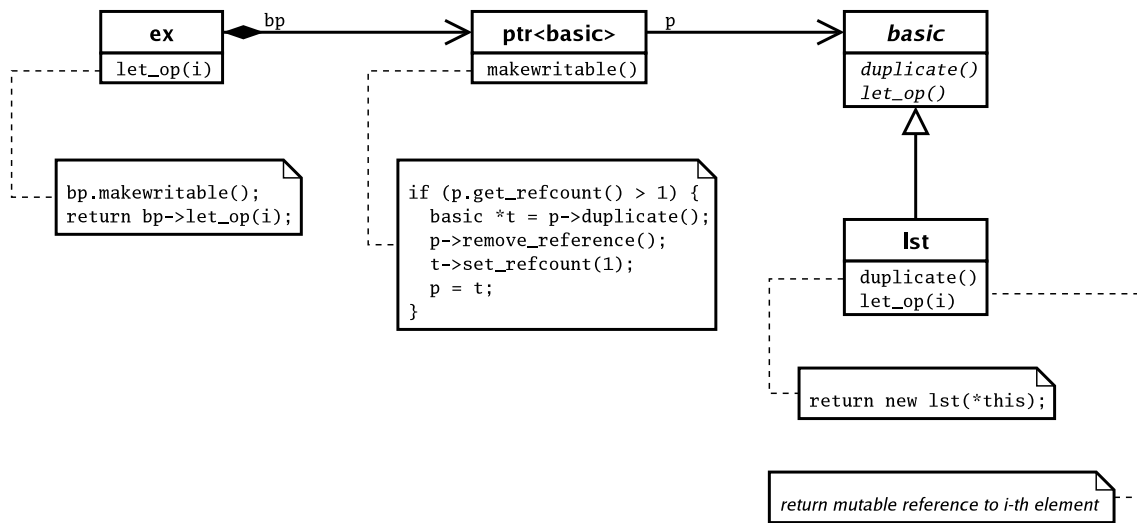


Figure 2.7: Copy-on-write: Attempting to modify an object that is referenced by more than one expression will create an exclusive duplicate first.

2.9.2 Copy-on-Write

The reference counting in GiNaC works well because expressions in GiNaC are generally non-mutable, i.e. the attributes and operands of algebraic objects can't be changed once they have been created. It is only possible to generate modified copies of expressions.

The `lst` (list) and `matrix` classes are the only two exceptions. List elements can be inserted or removed, and matrix elements changed without having to create new objects. This poses a problem for sharing objects between expressions because it is usually not desired that changing an object in one expression also modifies other expressions that happen to reference that same object.

The solution is called *copy-on-write*: When an object that is about to be modified (written to) is referenced by more than one `ex` instance, it is cloned (copied) first, and the modification carried out in the copy only. This is realized in a method `makewritable()` which is called by every `ex` member function that potentially modifies its referenced object, and a polymorphic cloning function `duplicate()` defined by `basic` and implemented by each algebraic class (see fig. 2.7).

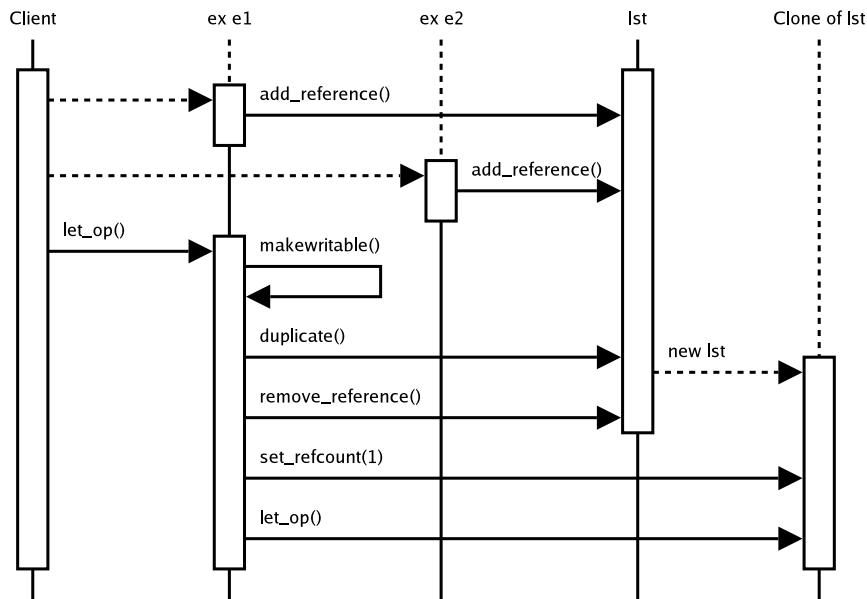
One expression-modifying method is `let_op(i)` which lets users modify list elements (as opposed to `op()` which only provides read-only access):

```

1 ex e1 = lst(1, 2, 3);
2 ex e2 = e1;
3 e1.let_op(1) = 0;

```

`e1` is first initialized with the list `{1, 2, 3}`, and after line 2 `e2` will point to the same object, as described. Calling `let_op()` in line 3 will make a copy of that list, change the second element, remove `e1`'s reference to the original list, and make `e1` point to the modified list. The original list is now referenced by `e2` only, which remains `{1, 2, 3}` while `e1` is now `{1, 0, 3}`:



2.9.3 Cloning of Stack-Allocated Objects

Memory management in GiNaC is further complicated by the fact that C++ actually *does* have one kind of automatically managed storage space: the stack on which local variables and temporary objects are allocated.

Objects on the stack must of course not be deleted when the last expression referring to them goes away. In fact, they can't reasonably be referenced by `ex` objects at all because this will leave dangling references when they are deallocated by C++.

Consider the following function which returns a newly created list:

```

1  ex f()
2  {
3      return lst(1, 2, 3);
4  }

```

In this case, the `lst` is an anonymous temporary object that is deleted when the function `f()` exits. If the returned `ex` contained a reference to that object, it would point to a no longer accessible memory location after returning from the function.

GiNaC handles this by asserting that `ex` instances always point to *heap-allocated* objects. If an expression is created from a stack-allocated object, GiNaC will first make a copy of that object (using the same `duplicate()` method that is used by the copy-on-write mechanism) and initialize the `ex` with a reference to the copy.

Thus, in a simple sequence such as

```

1  symbol x("x");
2  ex e = x;

```

GiNaC actually first clones the symbol `x` on the heap, and then makes `e` point to the clone which is now completely under its ownership. Even if the variable `x` goes out of scope, the

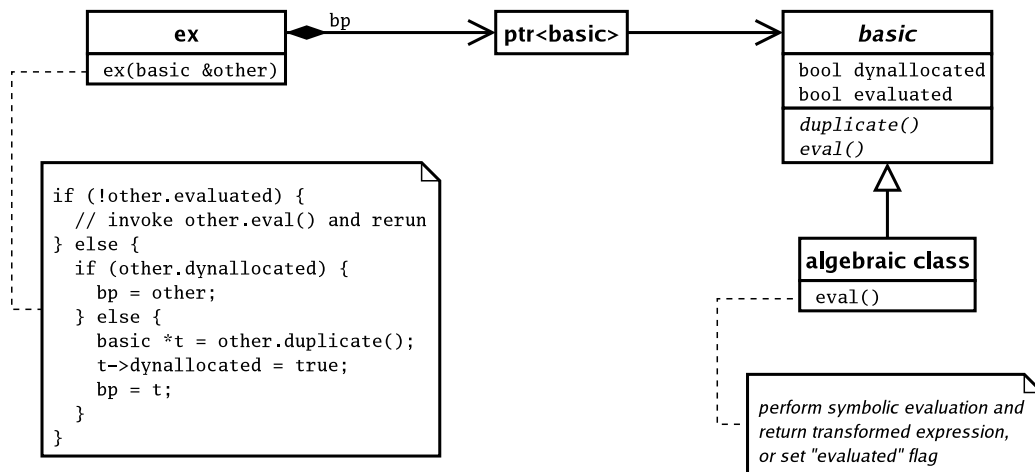
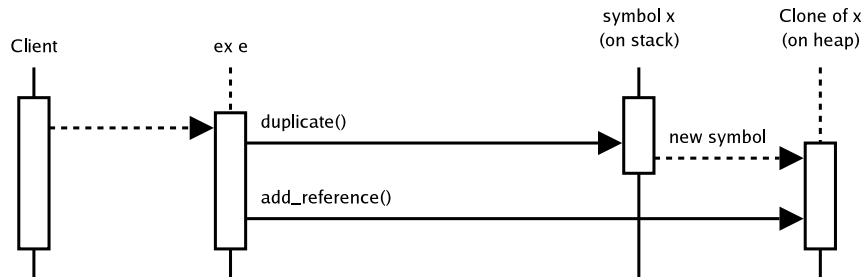


Figure 2.8: Binding a stack-allocated algebraic object (either a local variable or a temporary object) to an expression creates a dynamically allocated copy. At the same time, GiNaC performs an automatic symbolic evaluation.

expression `e` still references a valid symbol (which in turn gets deleted when `e` is destroyed or reassigned).



Unfortunately, in C++ it is not possible in a portable fashion to automatically determine whether an object lives on the stack or on the heap; some manual assistance is required. In GiNaC, all algebraic objects contain a flag `dynallocated` that needs to be set for all heap-allocated objects (those created with the `new` operator). When an object is bound to an `ex`, this flag is checked and the object cloned if the flag is not set (see fig. 2.8). The flag is then of course set in the clone.

Having to manually set this flag for every dynamically allocated object may sound cumbersome (and it is, to some extent), but it is normally not necessary for the user to explicitly allocate algebraic objects with the `new` operator. Most dynamic memory allocation happens inside the GiNaC library itself as part of the automatic memory management.

2.9.4 Flyweights

Sharing objects between expressions via the reference counting mechanism only works for objects that already exist. Keeping the number of newly-created objects down requires different strategies.

This problem is particularly obvious in the case of numbers. Every numeric literal that appears in a C++ expression that is in turn part of a GiNaC expression must first be converted to a `numeric` object before it can be part of the expression. For example, for an expression like $x^2 + 2x + 2$ which would be entered as

```
1 symbol x("x");
2 ex e = pow(x, 2) + 2*x + 2;
```

GiNaC would have to create three separate instances of `numeric` objects representing the number 2 that could easily be shared.

To mitigate this situation, GiNaC keeps around a pool of preallocated `numeric` objects for some commonly-used numbers. When one of these numbers is needed for an expression, a reference to the existing object is then returned, instead of creating a new one.⁸ This mechanism corresponds to the *Flyweight* pattern described in [GHJV 1995]. The constructors of `ex` from built-in C++ integral types (which are also called for implicit conversions to `ex`) assume the role of flyweight factories.

The expression resulting from the above code fragment thus actually only contains three references to one preallocated `numeric` object for the number 2.

2.9.5 Fusion of Redundant Subtrees

As a last measure to reduce the number of objects in expressions, GiNaC 1.2 implements the idea of *defensive fusion* presented in [Krec 2002, section 3.2].

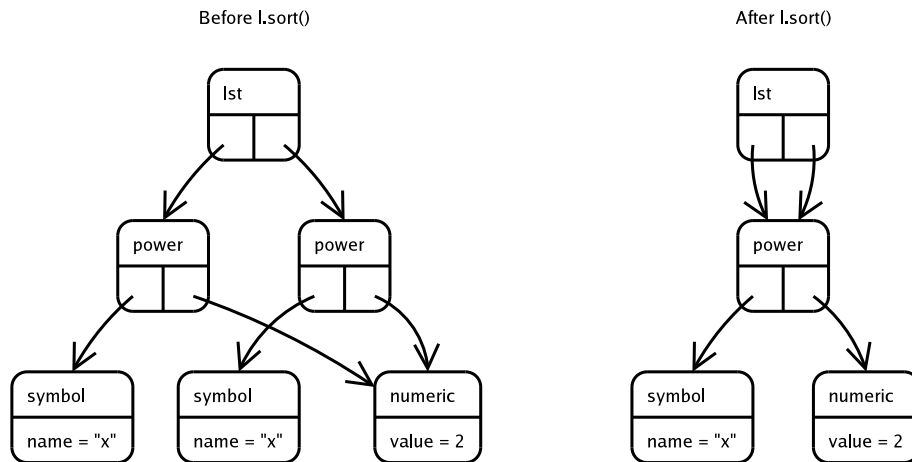
When during the comparison of two (sub)expressions they are found to be equal but are actually two separate subtrees, GiNaC will change the reference to one of the trees (the one with the lower reference count) to point to the other one, thereby possibly deleting one redundant subtree in the process (when it is no longer referenced by any other expressions).

As an example, consider the following code sequence which creates a list containing $2x$ twice, as different objects, and then sorts it:

```
1 symbol x("x");
2 lst l(pow(x, 2), pow(x, 2));
3 l.sort();
```

Creating the list in this fashion will not only construct two different `power` objects but it will also make two distinct clones of the symbol x (only the flyweight `numeric` 2 is shared). When the list is being sorted in line 3, the two elements will be compared, found to be equal, and one of the two redundant expression trees will be deleted:

⁸ An alternative approach would have been to set up a cache for recently-used numbers that is consulted every time a new `numeric` is about to be created.



Whether the subtree fusion yields any advantages is highly dependent on the type of calculation being performed. Considering the standard GiNaC 1.2 regression test and benchmark suite, the total memory savings as well as the reduction in the number of allocated objects are well below 1%. Of course, the benchmark suite consists of many independent routines between whom no objects are shared (with the exception of flyweights). Longer, more continuous calculations such as those performed in [BKK 2002] show better effects (approx. 15% reduction in memory usage, according to [Krec 2002]).

In any case, the main benefit of the method comes from the decrease in run-time (up to 8% for some of the GiNaC benchmark tests) by cutting down the number of required explicit comparisons between subtrees by nearly one-half (cf. section 2.11 below).

2.10 Run-Time Type Information

While an ideal object-oriented design is supposed to hide implementation-specific details of different classes behind abstract interfaces, it is occasionally necessary to determine the dynamic types of the objects contained in an expression.

C++ supports run-time information on types via the `typeid` operator and the associated `type_info` class. For reasons of efficiency, however, GiNaC implements its own lightweight run-time type information (RTTI) system based on integer type ID values stored directly in each algebraic object in a member variable `tinfo_key`. With the exception of the `structure<T>` template (see section 2.13), the actual ID values are predefined constants, one for each class, which makes comparisons of type IDs very fast as they are known at compile time.

On the downside, such hard-coded ID values pose a problem for users extending the library with new classes, because a value used for one class may already be in use for a different class from another developer. Moreover, it is the responsibility of each class to correctly set its `tinfo_key` attribute upon construction. Lastly, this scheme only allows determining the exact type of an object, without respect for class inheritance.

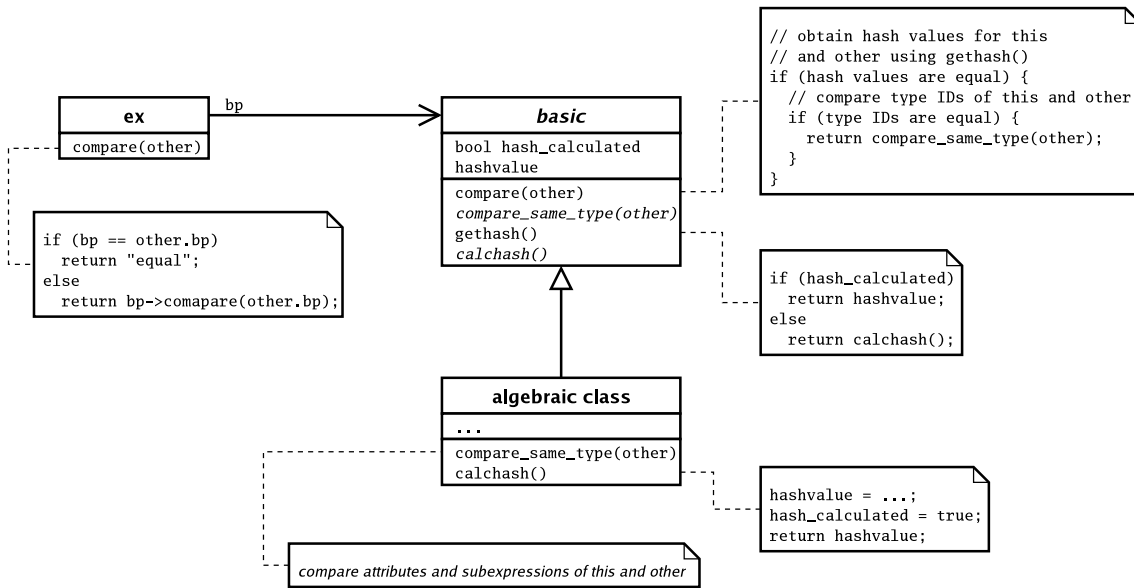


Figure 2.9: Comparing expressions relies largely on hash values that are cached inside algebraic objects.

GiNaC thus provides two functions for determining the dynamic type of expressions:

- `is_exactly_a<T>(e)` uses the described custom type ID mechanism to check whether the object pointed to by the expression `e` is an instance of the class `T`,
- `is_a<T>(e)` relies on the slower `dynamic_cast` operator to test whether the referenced object is an instance of `T` or one of its subclasses.

For example, `is_a<exprseq>(sin(x))` would return `true` (because `function` is a subclass of `exprseq`) while `is_exactly_a<exprseq>(sin(x))` would yield `false`.

The type ID values in algebraic objects are also employed for the table look-ups in GiNaC's implementation of double dispatch described in section 2.14, for specifying the commutation properties of objects (see section 3.4), and for the calculation of hash values for expressions, which will be explained in the next section.

2.11 Comparing Expressions

Profiling typical GiNaC applications reveals that up to 30% of their execution time is spent on comparing expressions, whether it be in the sorting of terms during the canonicalization of sums and products or by direct comparisons, usually with some fixed values like 0 or 1. It is therefore imperative that comparisons be as fast as possible.

The principal function of the `ex::compare()` method is to establish a canonical ordering relation on expression objects.⁹ Comparisons between two expressions are carried out in

⁹ As opposed to, say, the arithmetic ordering of real numbers. Even for arbitrary expressions a and b for which the truth of the relation $a > b$ is undetermined, GiNaC still needs to be able to put them into a canonical order when sorting expression vectors.

four steps:

1. If both expressions reference the same object, they are immediately known to be equal. This is why effective sharing of objects between expressions also helps speeding up comparisons.
2. If the objects are different, GiNaC computes a *hash value* for each expression, typically using the referenced object's type ID value and the hash values of any subexpressions the object may have (see [Frin 2000, section 2.2.4] for details about the computation and the advantages of using hash values). Once computed, the hash value is cached inside the object in a member variable `hashvalue` and a flag `hash_calculated` is set to indicate that a hash value for that object has already been calculated and can be used in subsequent comparisons without the need to recompute it (see fig. 2.9).¹⁰
3. If the hash values match, GiNaC compares the numeric type ID values stored in the referenced objects.
4. If the type IDs are the same, both objects are known to be instances of the same class and thus have the same set of attributes. GiNaC then invokes the method `compare_same_type()` which is implemented by each algebraic class to individually compare all attributes and subexpressions of the two objects (which are now known to belong to the exact same class).

Since it is often sufficient to compare two expressions for equality, without determining an order between them, GiNaC also provides a method `ex::is_equal()` (with an associated `is_equal_same_type()`) that works similar to `ex::compare()` but is often much faster. Algebraic classes are not required to implement `is_equal_same_type()` since this operation falls back to `compare_same_type()` by default.

Experiments show that, running the standard GiNaC benchmark suite, about 14% of all comparisons are settled by comparing the expressions' object pointers (step 1 above), and another 79% are reduced to comparing hash values (step 2). Only in the remaining 7% of all comparisons do algebraic objects need to be compared individually (without object fusion, this number nearly doubles to 13%).

Also, in 93% of the cases a hash value for an object is needed, it is taken from the cache without having to be recomputed.

2.12 Class Registry

GiNaC maintains a run-time database of all defined algebraic classes called the *class registry*. This database is set up at library initialization time via static member objects defined in each class, and is mostly used for internal functions of GiNaC.

The registry stores the following data for each class which can be looked up specifying a class name (see fig. 2.10):

¹⁰ Any method such as `let_op()` that modifies an algebraic object must, of course, clear this flag.

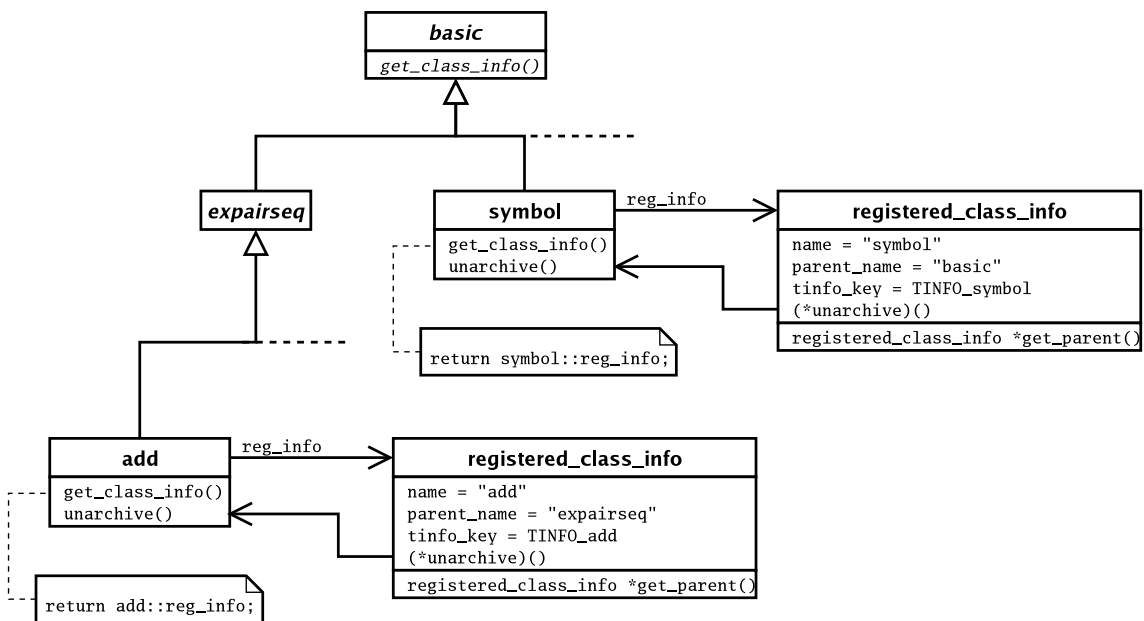


Figure 2.10: GiNaC maintains a registry of all defined algebraic classes. There exists one static `registered_class_info` object per class storing class-specific data such as the name of the class and its superclass, the type ID, and a pointer to an unarchiving function.

- the names of the class and its superclass as strings,
- the type ID for the class,
- a pointer to the `unarchive()` method that is used for restoring expressions from archives (see section 2.15),¹¹
- the method table for the `print()` double dispatch mechanism (see section 2.14).

A similar database is maintained for the hierarchy of `print_context` classes that play a role in specifying expression output formats, which will also be considered in section 2.14.

2.13 Adding New Algebraic Classes

A slightly awkward consequence of GiNaC’s type information system and class registry is that defining a new algebraic class (which, as outlined in the development goals, has to be possible without recompiling the library) is a little more involved than just deriving a new class from `basic`. To simplify the process, GiNaC provides three preprocessor macros:

- `GINAC_DECLARE_REGISTERED_CLASS(name, super)`

This macro is invoked in the first line of a class definition. It defines

1. the static `reg_info` object and associated helper access methods such as `get_class_info()` for the class registry,

¹¹ Having a way to obtain the `unarchive()` method of a class from its name was in fact the original reason for the implementation of the class registry.

2. the `duplicate()` method for cloning objects, and
3. the nested `visitor` class and the `accept()` method for the visitor mechanism (see section 2.18).

It also declares

1. the default constructor,
 2. the constructor and the `archive()` and `unarchive()` methods used by GiNaC's object persistence model (see section 2.15), and
 3. the `compare_same_type()` method.
- `GINAC_IMPLEMENT_REGISTERED_CLASS(name, super)` and `GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(name, super, options)`

One of these two macros usually appears in the source file that contains the class implementation, and defines the static `reg_info` object. The second form allows specifying additional class options which are stored in the registry, such as specifications of printing methods (see section 2.14).

Additionally, the programmer must pick an integer constant to be used as the type ID for the new class, which must follow the naming scheme `TINFO_classname`.

A minimal example for the definition of a new algebraic class `foo` would then look like this:

In a file `foo.h`:

```

1  const unsigned TINFO_foo = 0x42420001;
2
3  class foo : public basic {
4      GINAC_DECLARE_REGISTERED_CLASS(foo, basic)
5
6  public:
7      // any additional constructors and methods
8
9  private:
10     // any class attributes
11 };

```

In a file `foo.cpp`:

```

1  GINAC_IMPLEMENT_REGISTERED_CLASS(foo, basic)
2
3  // default constructor
4  foo::foo() : basic(TINFO_foo) {}
5
6  // archiving method
7  void foo::archive(archive_node &n) const
8  {
9      basic::archive(n);
10     // store class attributes in archive_node
11 }
12
13 // unarchiving constructor
14 foo::foo(const archive_node &n, lst &sym_lst) : basic(n, sym_lst)

```

```
15 {
16     // retrieve attributes from archive_node
17 }
18
19 // unarchiving method
20 ex foo::unarchive(const archive_node &n, lst &sym_lst)
21 {
22     return (new foo(n, sym_lst))->setflag(status_flags::dynallocated);
23 }
24
25 // comparison method
26 int foo::compare_same_type(const basic &other) const
27 {
28     // compare attributes of *this and other
29     ...
30 }
```

This represents the minimal set of methods that must be implemented by an algebraic class in GiNaC. Any useful class will, of course, also override additional methods from `basic`, like `eval()` and `calchash()`, and provide new methods for accessing class-specific attributes and functions.

To further simplify the creation of classes for users of the library, GiNaC 1.2 provides a template class `structure<T>` that wraps any C++ class `T` into an algebraic GiNaC class, as long as `T` has value semantics (i. e. variables of type `T` can be assigned and passed to and from functions like any built-in C++ data type). To make expression comparisons work, the user must also provide `==` and `<` operators for `T`. Any `basic` methods of `structure<T>` can be overridden via template specialization. The type ID for the new class is dynamically allocated from a range of reserved values.

A definition of `foo` using the `structure` template might look like this:

```
1 struct foo_T {
2     foo_T() {}
3
4     // any class attributes
5 };
6
7 bool operator==(const foo_T &lhs, const foo_T &rhs)
8 {
9     // determine whether lhs == rhs
10 }
11
12 bool operator<(const foo_T &lhs, const foo_T &rhs)
13 {
14     // determine whether lhs < rhs
15 }
16
17 typedef structure<foo_T, compare_std_less> foo;
```

The `=>` policy class parameter `compare_std_less` informs the `structure` template that the `==` and `<` operators are available for the `foo_T` class. It can be omitted, but then GiNaC would treat all `foo` objects as identical.

The observant reader will have noticed that in the examples given there are no methods defined for expression output. While there is a `print()` method in the `basic` class that

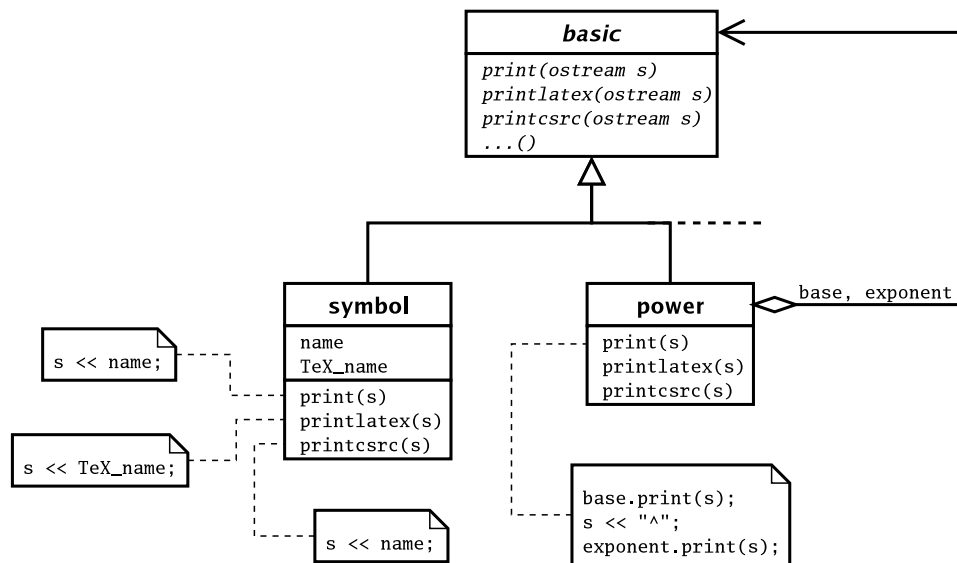


Figure 2.11: Prior to GiNaC 0.8.1, each expression output format was implemented as a separate method.

can be overridden by subclasses to implement the specific way in which an object should print itself, the preferred way to specify output formats in GiNaC is a much different one, and will be explained in the next section.

2.14 Expression Output

GiNaC allows printing expressions in a variety of different formats, including

- a MAPLE-like textual form (e. g. $1/4*\text{abs}(x)*y^2$),
- L^AT_EX source code (e. g. $\frac{1}{4} \{|x|} y^{\{2\}}$),
- C source code for numeric evaluation (e. g. $(1.0/4.0)*\text{fabs}(x)*(y*y)$).
- a “raw expression tree” text format for debugging.

It also permits defining new output formats and modifying existing ones.

The implementation of expression output in GiNaC went through several iterations. In versions of GiNaC up to and including 0.8.0, it was done in the straightforward object-oriented way by defining virtual methods like `print()`, `printlatex()`¹², `printcsrc()`, etc. in the `basic` class, which were then implemented by each algebraic class to provide the output in the specific way required for each class (see fig. 2.11), so in effect each object knew how to print itself in the various formats.

This worked reasonably well but had the critical disadvantage that creating a new output format, even if it was just a slight modification of an existing one, required adding a new

¹² L^AT_EX output didn’t actually exist in GiNaC 0.8.0 but was being considered at the time, which eventually led to the reimplementing of expression output.

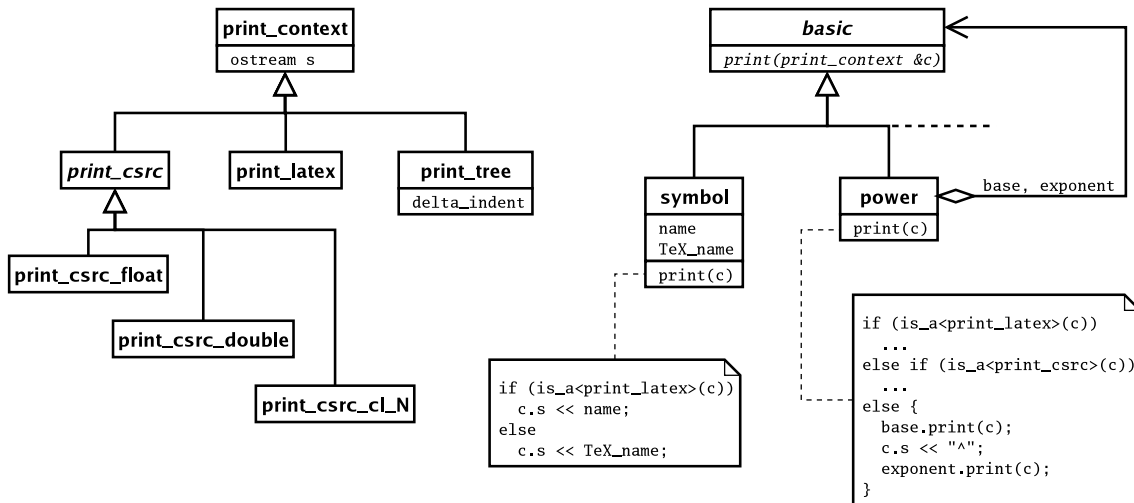


Figure 2.12: In GiNaC 0.8.1 and later versions, the output format is selected in a single `print()` method based on the dynamic type of an object belonging to the `print_context` hierarchy.

`print*()` method to `basic` and then implementing that method in *every single* algebraic class. It was also somewhat awkward to have to call a different method to get a different output representation.¹³

To improve upon this situation, GiNaC 0.8.1 introduced a hierarchy of classes corresponding to the different output formats, rooted in a class named `print_context`, and replacing the various `print*()` methods with a single `print()` that takes a `print_context` object as an argument (see fig. 2.12). The `print_context` classes don't implement any logic themselves but only serve for selecting a specific format inside the `print()` implementations of the algebraic classes. In addition, they can also store any contextual state information that a specific output format might need, such as the indentation level in the “tree” format.

While this implementation neatly expressed the relationships between the different output formats and had the advantage of providing some sort of fallback mechanism (for example, `numeric` is the only class that needs to distinguish between the different subclasses of `print_csrc`, so adding a new format that is a subclass of `print_csrc` only requires changing `numeric::print()`), in effect it only converted the various `print*()` methods into conditional statements inside the `print()` implementations of all classes. Adding a new printing format was slightly easier as it didn't change the interface of the `basic` class, but it was still not possible without recompiling the library.

Using this interface, printing an expression in \LaTeX format requires writing code like

```

1 ex e = ...
2 e.print(print_latex(cout));

```

¹³ In anticipation of section 2.18 we should note that applying the Visitor pattern to the `print()` methods would solve the problem of adding new output formats (simply define a new visitor class for the new format) but would complicate the creation of new algebraic classes since the visitors for the existing formats would have to be extended by a new `visit()` method for the new class.

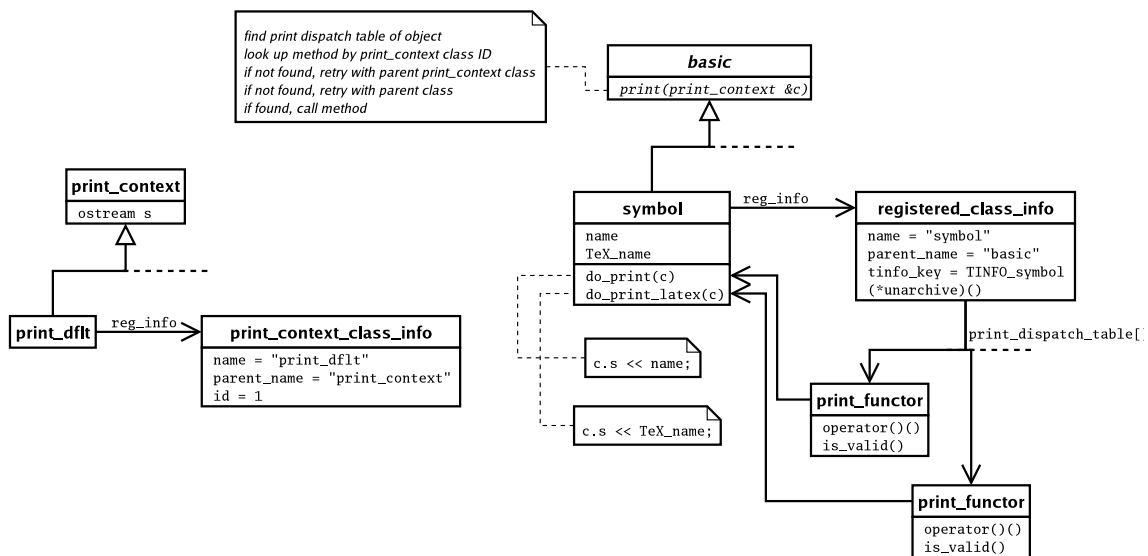


Figure 2.13: Double dispatch for `basic::print()` in GiNaC 1.2. The GiNaC class registry is used to obtain a class-specific method table from which a function is selected according to the dynamic type of the passed `print_context` object. Information about the `print_context` hierarchy is maintained in a separate class registry. The method tables can be changed at run time.

GiNaC 1.1 made this slightly more convenient by providing stream manipulators for all built-in formats which allows writing this in a more C++-like fashion:

```

1 ex e = ...
2 cout << latex << e;
  
```

Having the output format selected by a `print_context` class proved advantageous for this since the `<<` operator for expressions only needs to call the expression's `print()` method passing a `print_context` object that is created by one of the manipulators.

Given the mentioned disadvantages, a complete rewrite of the expression output system was undertaken for GiNaC 1.2. Upon closer examination it is apparent that what is really needed here is a \Rightarrow double dispatch mechanism, i. e. the ability to determine which method to call given the dynamic type of *two* objects, one belonging to the `basic` hierarchy and the other one belonging to the `print_context` hierarchy. The C++ language only supports single dispatch (via its virtual function feature), so some manual work is needed.

Logically, the printing methods are stored in an asymmetric matrix of (possibly empty) references to functions which is initialized on startup but can also be manipulated at run time. Physically, the GiNaC class registry maintains for each algebraic class a dispatch table of function references corresponding to the different `print_context` types, as illustrated in figs. 2.13 and 2.14. These tables are indexed by type ID values which are dynamically allocated for the `print_context` classes. For this purpose, there is a second class registry for the `print_context` hierarchy, similar to the one for algebraic classes, that also stores information about the inheritance relationships of the classes. Both registries use templates to share common code and data structures between them.

The entries in the dispatch tables can be pointers to member functions, pointers to global functions, or null pointers representing unspecified methods. The syntactic differences in

	print_context	print_dflt	print_csrc	print_csrc_cl_N	print_latex	...
basic	●	←	←	←	←	
power	↑	●	●	←	●	
add	●	←	●	←	●	
mul	●	←	●	←	●	
symbol	●	←	←	←	●	
numeric	●	←	●	●	●	
idx	●	←	←	←	●	
varidx	●	←	←	←	↑	
...						

Figure 2.14: The output methods of algebraic objects are arranged in a matrix. For every combination of object class and output format a handler method can be specified (●). Unassigned methods fall back to assigned methods in the same class (←) or a superclass (↑).

the specification and invocation of these are encapsulated by the class `print_func`.

`basic::print()` first obtains a pointer to the printing dispatch table of the actual dynamic class of the object it is invoked on. Next it looks up the method in the table using the type ID of the supplied `print_context` object obtained from the `print_context` class registry. If a method has been specified it is called, otherwise GiNaC determines the type ID of the `print_context` object’s superclass and consults the dispatch table again, etc. until either a method is found or it arrives at the base of the `print_context` class hierarchy. If this is the case, `basic::print()` gets a pointer to the dispatch table of the algebraic object’s superclass and retries the process. This provides the desired fallback behavior for the method dispatch while still allowing (best-case) constant-time dispatch, and also distinguishes the GiNaC implementation from, for example, the multiple-dispatch techniques presented in [Alex 2001, chapter 11].

The printing methods for an algebraic class are specified via the `options` parameter of the `GINAC_IMPLEMENT_REGISTERED_CLASS_OPT` macro. As described, GiNaC allows passing member functions as well as global functions. In the latter case, the first parameter of the function will be a pointer to the algebraic object. Both types of functions also receive a reference to the `print_context` object of the original `print()` request. Templates are used to allow some flexibility in the argument types of the functions. For example, the method called for \LaTeX output can accept a reference to either a `print_latex` or a `print_context` object.

Printing methods can also be changed at run-time by calling the global function `set_print_func()`. This works for both newly defined algebraic classes as well as built-in classes provided by GiNaC.

To exemplify, the `power` class implementation specifies its output methods like this:

```

1 GINAC_IMPLEMENT_REGISTERED_CLASS_OPT(power, basic,
2   print_func<print_dflt>(&power::do_print_dflt).
3   print_func<print_latex>(&power::do_print_latex).
4   print_func<print_csrc>(&power::do_print_csrc))

```

where `do_print_dflt()`, `do_print_latex()`, and `do_print_csrc()` are member functions of `power`. When `print()` is invoked on a `power` object with a `print_latex`

context GiNaC will call `power::do_print_latex()`, when it is invoked with a `print_csrc_double` or `print_csrc_cl_N` context it will call `power::do_print_csrc()`, and with a `print_tree` context it will fall back to the `do_print_tree()` method defined by the `basic` class.

A GiNaC user can also define a new printing context class, say `print_fortran`, and use `set_print_func()` to specify how a `power` object should print itself in this format. The way of defining `print_context` classes is similar to that of algebraic classes, relying on `GINAC_DECLARE_PRINT_CONTEXT` and `GINAC_IMPLEMENT_PRINT_CONTEXT` macros.

GiNaC's expression output mechanism thus achieves its three main goals:

1. it provides an efficient and easy-to-use implementation of double dispatch in which methods can be assigned both at compile time and at run time, and that features an inheritance system similar to ordinary virtual function dispatch in C++,
2. new `print_context` classes for user-defined output formats can be easily added without recompiling the library, and
3. the ability to add new algebraic classes is unaffected.

Essentially the same mechanism is used for symbolic functions to allow the specification of different output formats for them. For example, the absolute value function prints itself as `abs(x)` in the default format, as `fabs(x)` in the C source format, and as `|x|` in the \LaTeX format. The method dispatch tables of functions are kept in the associated `function_options` objects, and `function` overrides `basic::print()` (it is the only algebraic class to do so) to perform the lookup and dispatch.

2.15 Object Persistence

For some applications it is useful to be able to store GiNaC expressions in a persistent location such as a file on disk so they can be read back at a later time or by a different program.

While it is possible to output expressions in text format and read them back using GiNaC's input parser, this approach has a couple of drawbacks:

- Not all attributes of GiNaC objects appear explicitly in the output and would thus become lost in the process. For example, commutation properties of symbols and the dimensionality of indices are hidden in the default output format.
- The input parser reconstructs expressions from scratch, undoing the graph structure and any shared subexpressions the original object tree might have had.
- Storing more than one expression in the same file requires some manual parsing to discern the individual expressions.

GiNaC implements a more advanced method of achieving object persistence, referred to as *archiving*.¹⁴ One or more expressions can be stored in an `archive` object which converts them to a sequence of `archive_nodes`, each one holding the attributes and sequence indices of subexpressions of one algebraic object as property-value pairs.¹⁵ The contents of an `archive` can then be written to a data stream (such as a file or a network connection) in a platform-independent, binary format.

To retrieve an expression, the file is read back into an `archive` object, and the expression's object graph reconstructed from the stored `archive_nodes`. The mechanism preserves the expression graph structure even for objects shared between multiple expressions, as long as they are stored in the same `archive` object.

Internally, the archiving subsystem relies on the two methods `archive()` and `unarchive()` implemented by each algebraic class to store and retrieve all object attributes to and from an `archive_node`, respectively. An object's class is encoded by storing the class name as a string, because the names were deemed to be more long-lasting than the type IDs. Upon unarchiving, GiNaC retrieves a pointer to the `unarchive()` function of a class via its name, by looking it up in the class registry (see above).¹⁶ This part of the archiving system thus acts as an object factory.

Symbols and numeric values (i.e. values of `numeric` objects) are also stored as strings (symbols by their name, and `numerics` in a special integer-decoded format that preserves all information about a number). For efficiency reasons, all strings in archives are represented by unique ID numbers, a concept that is inspired by the X Window System's use of *atoms* [Gett 1989].

2.16 GiNaC Expressions and the C++ Standard Library

With versions 1.1 and 1.2, increasing importance was attached to having GiNaC interoperate seamlessly with the C++ Standard Template Library (STL). The three main groups of components provided by the STL are

1. container classes,
2. generic algorithms,
3. iterators connecting the algorithms with the containers.

This section focuses on STL containers. The use of iterators in GiNaC will be discussed in the next section.

Storing GiNaC expressions, i.e. objects of type `ex`, in STL containers is mostly straightforward, as they have value semantics; expressions can be copied, assigned, etc. like any

¹⁴ The terminology and parts of the implementation concept are taken from the object persistence model of the Be Operating System [Be 1997, chapter 6].

¹⁵ In the language of [GHJV 1995], an `archive_node` acts as a *Memento* of an algebraic object.

¹⁶ The BeOS doesn't have a class registry; it obtains a pointer to the unarchiving function by accessing the application's symbol table directly. While this approach has been successfully tested with GiNaC (under Linux, using `dlopen()/dlsym()`), it was deemed too platform-specific for general use.

built-in type. However, *sorted associative* containers like `set` and `map` require that a *strict weak ordering relation* exists for the association keys. By default, STL containers will fall back to the C++ `<` operator to establish the ordering but, as explained, this operator is overloaded by GiNaC for expressions to return `relational` objects which do not define a strict weak ordering (in particular, they do not guarantee that exactly one of `a < b`, `a > b` and `a == b` is true).

Instead, GiNaC provides a comparison functor called `ex_is_less` that employs the `ex::compare()` method described in section 2.11 to establish a proper strict weak ordering on expressions. This functor has to be supplied to STL containers as a template parameter, like this:

```
1 map<ex, ex, ex_is_less> m;
```

The same `ex_is_less` functor must also be specified for STL algorithms that require a comparison function object, like `sort()` and `next_permutation()`. For example, to sort a `vector` of GiNaC expressions one would use

```
1 vector<ex> v;
2 // fill v with values...
3 sort(v.begin(), v.end(), ex_is_less());
```

To speed up STL algorithms, GiNaC overloads the STL `swap()` and `iter_swap()` functions for `ex` arguments. Two GiNaC expression objects can be swapped much more efficiently by just exchanging their `basic` pointers, than by the standard swapping algorithm using a temporary variable (`ex t = e1; e1 = e2; e2 = t;`) which involves some redundant reference counter manipulations. Measurements show that the overloaded swapping operations bring about a 10% speedup for an STL `sort()` of a `vector<ex>` with GCC 3.3.2.

2.17 Iterators

The purpose of iterators is to allow access to the elements of a container or a composite structure in a generic way without requiring knowledge of its underlying representation.

Since its very beginnings, GiNaC implemented the method `op(i)` to retrieve the $(i+1)$ -th subexpression of a container object such as a `sums` or a `list`. This method emulates the behavior of the `op(i,e)` function in MAPLE, and provides random element access.

There are, however, data structures that are not designed for particularly efficient random access. A prominent example is the doubly-linked list, such as implemented by the STL `list<T>` container, which only guarantees constant-time access to its first and (usually) last elements, and to any element's immediate predecessor and successor, but not to arbitrary elements.

This implies that the implementation of `op()` for GiNaC's list class `lst`, which is basically a `list<ex>`, requires on average $\frac{N}{2}$ steps to retrieve an element from a list of length N , which makes it an $\mathcal{O}(N)$ operation:

```
1 ex lst::op(size_t i) const
2 {
```

```
3     list<ex>::const_iterator p = seq.begin();
4     advance(p, i); // advance iterator by i elements
5     return *p;
6 }
```

But this in turn makes iterating over the elements of a list with `op()`, as in

```
1 lst l = ...
2 for (size_t i=0; i<l.nops(); ++i) {
3     // call l.op(i) which is O(N)
4     // ++i is O(1)
5 }
```

an operation of order $\mathcal{O}(N^2)$ because each of the N invocations of `l.op()` is itself of order $\mathcal{O}(N)$.

Given the commonness of iterating over lists in GiNaC applications and the library itself this of course constitutes a severe performance problem (in section 2.21 we will demonstrate the impact of this on substitution operations in GiNaC 1.0).

To improve upon this, GiNaC provided constant-time bidirectional iterators for lists starting with version 1.1, in the easiest possible way by simply exporting the STL `list<ex>::const_iterator` type via the `lst` class. Iterating over the elements of a list using this iterator then becomes a linear-time operation:

```
1 lst l = ...
2 for (lst::const_iterator i=l.begin(); i!=l.end(); ++i) {
3     // use *i which is O(1)
4     // ++i is also O(1)
5 }
```

Since the list iterators are ordinary STL iterators, they can also be used with all STL algorithms. For example, to add up the elements of a list one can use

```
1 lst l = ...
2 ex sum = std::accumulate(l.begin(), l.end(), ex(0));
```

Starting with GiNaC 1.2, iterators are available for all expressions, not just for lists. The class `const_iterator`¹⁷ wraps the `op(i)` interface into an STL compatible random access iterator, with the associated `ex::begin()` and `ex::end()` methods working in the usual way. For instance, to print the individual operands of a GiNaC expression, each on a line of its own, one could write

```
1 ex e = ...
2 std::copy(e.begin(), e.end(), std::ostream_iterator<ex>(cout, "\n"));
```

As `const_iterator` relies on `op()`, it shares with it the efficiency problems with lists (note that `lst::const_iterator` is a different class, taken directly from STL). It would of course have been possible to use polymorphism to allow algebraic classes to provide their own iterator subclasses suited to the individual data structures employed by them, but

¹⁷ Keeping in line with the general non-mutability of expressions, GiNaC's iterators only provide read access.

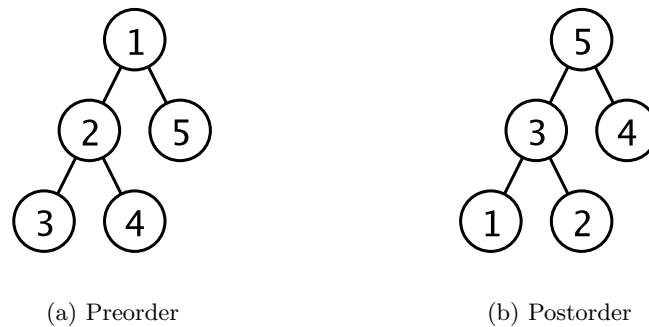


Figure 2.15: The order in which the nodes of a tree are visited during preorder (left) and postorder (right) traversal.

that would incur a performance penalty for all iterator operations. Since lists in GiNaC are usually not used as parts of complex expressions but rather as separate objects to manage a collection of other expressions, requiring the user to utilize a special iterator class for lists to get optimum efficiency was deemed an acceptable inconvenience.

One additional advantage of using iterators in an object-oriented design is that they can encapsulate different iteration algorithms instead of just allowing sequential element access. A common algorithm for composite structures like GiNaC expressions is the *tree traversal*. With `op()`, this typically leads to recursive functions like

```

1 void f(ex e)
2 {
3     // do something with e
4     // ...
5
6     for (size_t i=0; i<e.nops(); ++i)
7         f(e.op(i));
8 }

```

that throw the iteration logic together with the actual operation implemented by `f()`.

GiNaC 1.2 defines two special iterator types, `const_preorder_iterator` and `const_postorder_iterator`, that provide the two most common tree traversal algorithms (see fig. 2.15) with an interface compatible to STL forward iterators. For these types, the standard increment operator `++` actually performs a tree recursion instead of just advancing the iterator to the next operand. Internally, both iterator types use a stack to keep track of the current position inside the expression tree.

Using traversal iterators and the STL `for_each()` function, the code example above would be written as

```

1 void f(ex e)
2 {
3     // do something with e
4     // ...
5 }
6
7 std::for_each(e.preorder_begin(), e.preorder_end(), f);

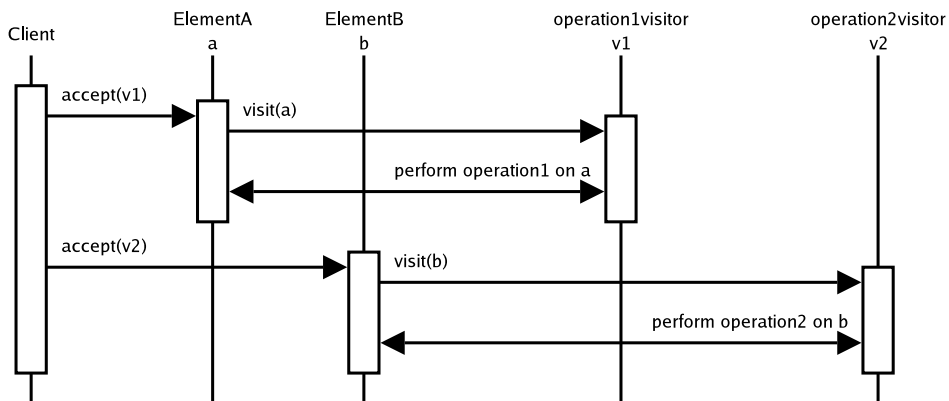
```

2.18 Visitors

In section 2.4 we exemplified how algorithms in an object-oriented design like GiNaC are usually expressed as a set of methods distributed over a class hierarchy and implemented in an appropriate way for each individual class.

Since, however, classes in C++ can't be augmented by additional methods once they are defined, a user of GiNaC can't actually write his own symbolic algorithms in this fashion, because he can't add new methods to the GiNaC classes. Class-specific behavior in an application using GiNaC would thus have to be implemented with type switches, which is exactly the kind of construct that the object-oriented programming paradigm aims to avoid.

The solution to this dilemma comes in the form of the *Visitor* pattern [GHJV 1995]. In this design, instead of representing operations as individual methods on the classes of an element hierarchy (in the case of GiNaC, the algebraic class hierarchy), the classes provide only one generic `accept(v)` method taking a `visitor` object as its argument. `visitor` is an abstract class defining exactly one `visit()` method for each element class. The `accept()` operations in the element classes invoke their associated `visit()` methods in the visitor. Subclasses of `visitor` override the `visit()` methods to provide the desired class-specific behavior (see fig. 2.16):



The Visitor pattern thus effectively pulls methods out of the element class hierarchy and puts them into a separate visitor hierarchy, now grouped by function. To add a new operation to the element hierarchy, one defines a new subclass of `visitor` with appropriate implementations of the various `visit()` methods, without the need to change the interface of the element classes themselves.

This approach would work fine with GiNaC, were it not for the fact that one of the design goals of GiNaC is to also allow extending the hierarchy of algebraic classes. Adding a new algebraic class, however, would require changing the `visitor` interface to add a new `visit()` method for the new class. But this is precisely the problem we had in the first place.

A slight modification to the Visitor pattern, the *acyclic visitor* presented in [Mart 1998] solves this issue. This pattern avoids the dependency cycle of the original Visitor pattern

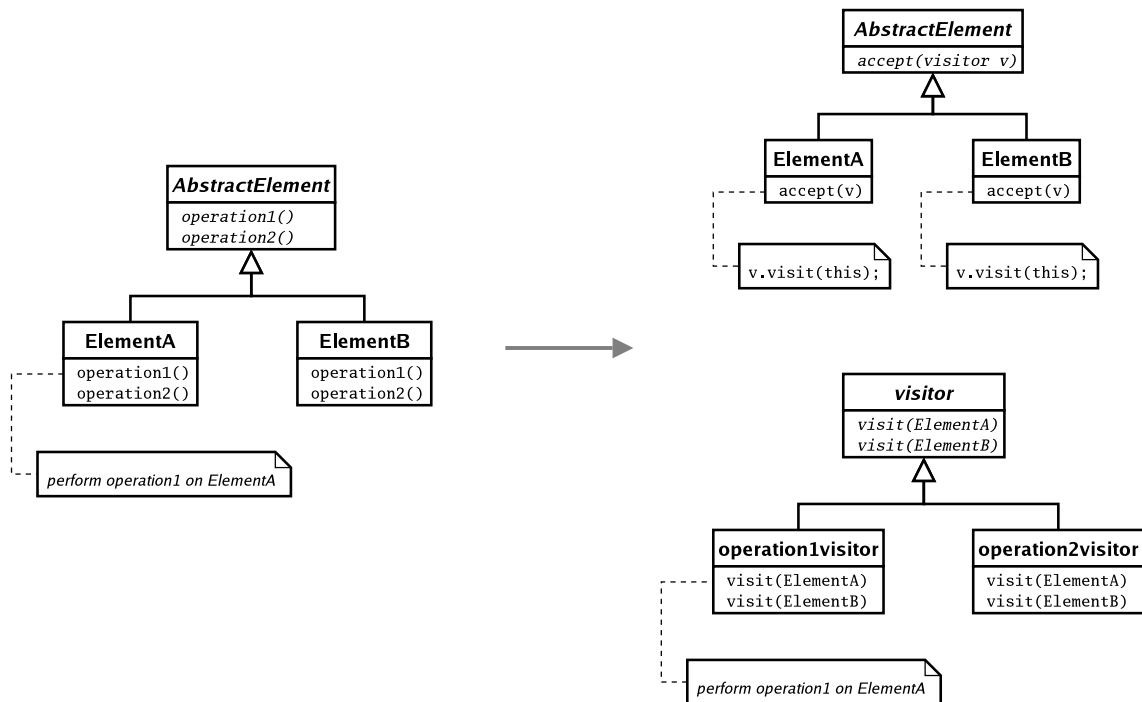


Figure 2.16: The Visitor pattern moves operations on the elements of a class hierarchy into a separate visitor hierarchy. Defining a new operation requires adding a new visitor subclass, but leaves the interface of the element classes unchanged.

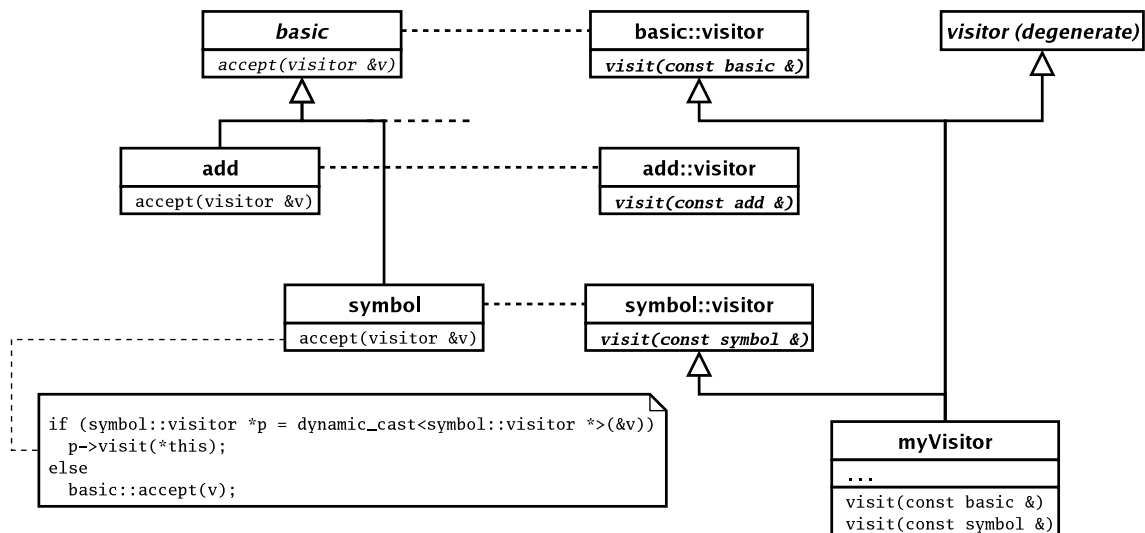


Figure 2.17: In the acyclic visitor pattern, a concrete visitor (`myVisitor`) inherits from the associated visitor classes of all the element classes it is interested in, as well as the global, degenerate visitor class. Adding a new element class does not require changes in the existing visitor classes.

(`visitor` depends on all element classes for the `visit()` methods, but at the same time all element classes depend on `visitor` for `accept()`) by defining separate visitor classes, one for each element type, with each one having only one single `visit()` method for its associated type. The `visitor` class that appears as the argument to `accept()` is a completely separate, *degenerate* class (i. e. one without any methods or attributes). A concrete visitor class inherits from all the element-specific visitor classes it is interested in, as well as the global, degenerate `visitor` (see fig. 2.17). The `accept()` method of each element class performs a cross-cast from `visitor` (which is the static type of the argument to `accept()`) to its specific visitor class (the one that declares the appropriate `visit()` method) and then invokes `visit()`.

In GiNaC, the element-specific visitor classes are nested inside the element classes themselves, and are automatically defined by the `GINAC_DECLARE_REGISTERED_CLASS` macro, along with the `accept()` method. In addition to that, GiNaC implements a simple method-propagation system for its visitors: If the cross-cast in `accept()` fails, i. e. if the actual visitor object passed has not been derived from the element-specific visitor class, GiNaC will forward the request to the element's superclass.

Visitor objects are often used to traverse all nodes of an expression. This can be achieved in a number of ways:¹⁸

- a) Manually recursing over subexpressions and calling `accept()` on each. GiNaC provides two methods, `ex::traverse_preorder()` and `ex::traverse_postorder()`, both taking a visitor object as an argument, that automate this process.
- b) Manually iterating over all subexpressions with a `preorder_iterator` or `postorder_iterator` and calling `accept()` on each.
- c) Putting the traversal algorithm inside the visitor by having the `visit()` methods of container classes call `accept()` on their subexpressions.

A visitor object can also store state which is accumulated during the expression traversal, and that would otherwise have to be passed around between method invocations on algebraic objects.

As a simple example, consider the definition of a visitor class that collects the set of all indices appearing in an expression, ignoring index variances so the contravariant $^{\mu}$ and the covariant $_{\mu}$ will be treated as the same index (see section 3.2.1 for more information about indices in GiNaC):

```
1 class gather_indices
2   : public visitor, public idx::visitor, public varidx::visitor {
3
4     void visit(const idx & x)
5     {
6         s.insert(x);
7     }
8
9     void visit(const varidx & x)
```

¹⁸ The `accept()` method itself does not automatically make a visitor visit an object's children. This allows more flexibility in the choice of the traversal algorithm.

```

10     {
11         s.insert(x.is_covariant() ? x : x.toggle_variance());
12     }
13
14 public:
15     set<ex, ex_is_less> s;
16 };

```

Inheriting from `idx::visitor` and `varidx::visitor` indicates that this visitor class has `visit()` operations for the `idx` and `varidx` index classes but no others. If the visitor encounters a `spinidx` which is a subclass of `varidx`, the `visit(varidx)` method will be called.

An instance of the `gather_indices` class can then be sent over an expression using an iterator or one of the `traverse()` methods:

```

17 ex e = ...
18 gather_indices v;
19 e.traverse_preorder(v);

```

The set of indices is then collected in the member variable `v.s`. Without visitors, this operation would typically be implemented using a recursive function and a type switch such as

```

1 void gather_indices(ex x, set<ex, ex_is_less> &s)
2 {
3     if (is_a<varidx>(x)) {
4         // insert contravariant version of index into s
5     } else if (is_a<idx>(x)) {
6         s.insert(x);
7     } else {
8         // recurse into subexpressions
9     }
10 }

```

which is prone to a subtle programming error: One has to check for the subclass (`varidx`) before the superclass (`idx`). When the order is reversed, the `if (is_a<idx>)` branch would also be executed for `varidx` instances.

The version of this algorithm using a visitor is about 20% slower than the recursive function, which is largely due to the expensive cross-cast operation in `accept()` (the C++ runtime system has to dynamically check whether the `visitor` object passed to `accept()` is also a subclass of either `idx::visitor` or `varidx::visitor`). But the performance of visitors gets better with larger numbers of element classes because each `visit()` has constant-time dispatch, whereas a type switch is effectively a linear search.

2.19 Applying Functions on Subexpressions

The visitor objects described in the last section are most useful for collecting information about expressions, or for extracting subexpressions matching some given criteria. For *transforming* expressions, especially in cases where the resulting expression is syntactically similar to the original one, GiNaC offers a mechanism that is very common in the functional

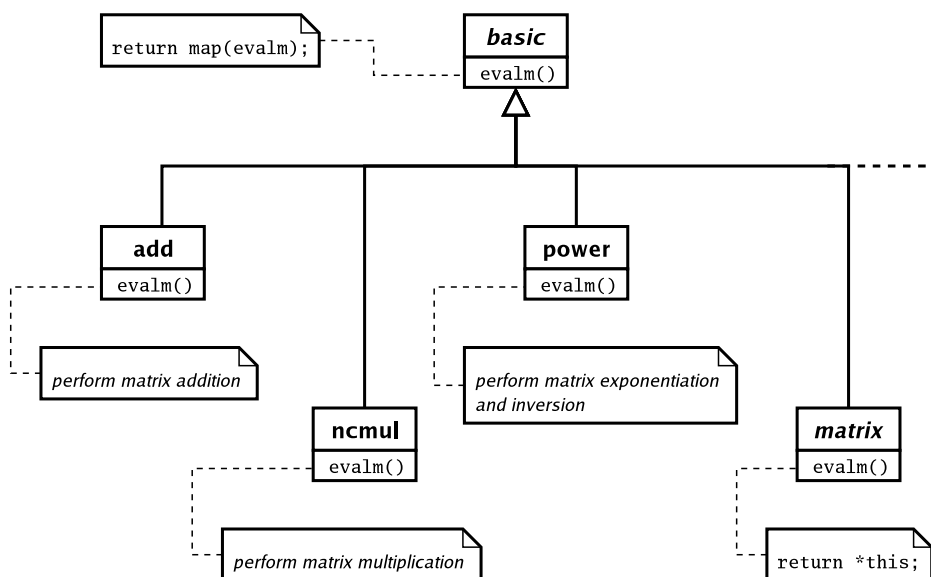


Figure 2.18: The default implementation of the matrix-evaluation method `evalm()` in the `basic` class uses `map()` to forward the operation to subexpressions.

paradigm: applying a function on all children of an object. For example, applying the function $f()$ on the expression $a + 2b - 1$ in this way would result in $f(a) + f(2b) + f(-1)$. Characteristic for this operation is that the type and attributes of the top-level object remain unchanged (in the example, both the original and resulting expressions are sums).

In GiNaC, a function is applied by calling the method `ex::map(f)`, where `f` is either a pointer to a C++ function taking and returning an expression, or a `map_function` object. `map_function` is a functor class that in some respects behaves similar to a visitor. It lacks the element type dispatch of the Visitor pattern but it also represents related operations in a single object and can accumulate state while traversing the nodes of an expression.

`map()` typically operates by duplicating the object it is called upon, and replacing its subexpressions with the transformed ones. Because they return modified expressions, `map_function` objects usually implement their own tree traversal.

GiNaC frequently uses `map()` internally to propagate methods to the elements of container classes, especially “passive” ones like `lst` that only store individual elements without defining any relation between them. One example of such an operation is `evalm()` which evaluates sums, products, and powers of matrices in expressions.¹⁹

Only the `add`, `mul`, `ncmul`, `power`, and `matrix` classes have special implementations of `evalm()`. The default behavior, defined in `basic::evalm()` is to map the `evalm()` method on the object’s subexpressions (see fig. 2.18) which is usually the desired behavior. In this way, invoking `evalm()` on a list will return a list of evaluated elements, invoking it on a relation will apply the evaluation to the expressions on both sides of the relation, etc. Note that the `matrix` class provides its own implementation of `evalm()` returning the unmodified matrix object. If it didn’t, `basic::evalm()` would invoke the evaluation on

¹⁹ Since these operations are of a complexity equal to or higher than $\mathcal{O}(N^2)$ for matrices of size N they are not carried out by GiNaC’s anonymous evaluator `eval()`.

all individual matrix elements, which is usually a waste of time since the elements of a matrix are rarely themselves matrices. But if required, the user can always manually map `evalm()` on a matrix to evaluate the elements.

An interesting application of `map()` is the implementation of *active* object fusion, i. e. combining all redundant subexpressions of an expression by searching through the entire object tree:

```

1  class map_fuser : public map_function {
2      set<ex, ex_is_less> s;
3
4  public:
5      ex operator()(const ex &e)
6      {
7          ex f = e.map(*this);
8          s.insert(f);
9          return f;
10     }
11 };
12
13 int main()
14 {
15     symbol x("x");
16     ex e1 = x/(1-x) + pow(1-x, 2);
17
18     map_fuser fuse;
19     ex e2 = fuse(e1);
20 }
```

`e1` contains three instances of the symbol x , and two instances of the sum $1 - x$. `e2` contains only one of each.

This implementation works because, as described in section 2.9.5, GiNaC fuses equivalent expression trees when comparing them, and inserting an expression into a `set<ex>` will result in a comparison with an existing equivalent expression if it is already in the set. The expression `f` returned from `map_fuser::operator()` will thus always refer to one of the objects that are already in the set of subexpressions `s`, as will any equivalent subexpressions encountered subsequently in the traversal.

Because the `fuse` object retains its state, it can be used multiple times to fuse the objects from different expressions.

2.20 Hash Maps

In GiNaC applications, and in GiNaC itself, expressions are frequently used as keys of associative containers such as the C++ Standard Library `set<ex>` and `map<ex, T>` types. These are *sorted* associative containers, i. e. they rely on an ordering relation between their keys, in this case the one provided by the `ex_is_less` functor, to guarantee logarithmic complexity (worst case) for inserting, erasing, and looking up elements.

⇒ Hash tables achieve (on average) constant complexity for these operations by storing the associated values in a table indexed by hash values computed for the keys. Since GiNaC already calculates hash values for its expressions it made sense to provide as a part of

GiNaC a generic hash table implementation, called a *hash map*, that acts as a drop-in replacement for the STL `map<>` type.²⁰

One of the key issues in implementing a hash table is how to deal with hash collisions, i. e. the case where two different keys have the same hash value and would thus point to the same table entry. Time-tested strategies include

- **Chaining:**

Resolve the ambiguity by making each *bucket* (table entry) contain not a single value, but a list of (key,value) pairs for all keys having the same hash value. A look-up operation searches the list to retrieve the correct value.

- **Probing:**

Each table entry contains only one value. When a collision happens the new value is stored in the next free table entry instead. There are different ways to determine a suitable “next” entry, such as *linear probing* (constant table index increments) and *quadratic probing* (linearly increasing increments). This is actually very similar to chaining, but it keeps the hash chains inside the table itself. Special care has to be taken when removing elements so as not to lose access to elements further down the chain.

- **Rehashing:**

Upon finding a collision, change the hash function (hopefully to a more optimized one), recompute all hash values, and recreate the table. Since the hash function for expressions in GiNaC is fixed, this is not an option.

- **Enlarging the table:**

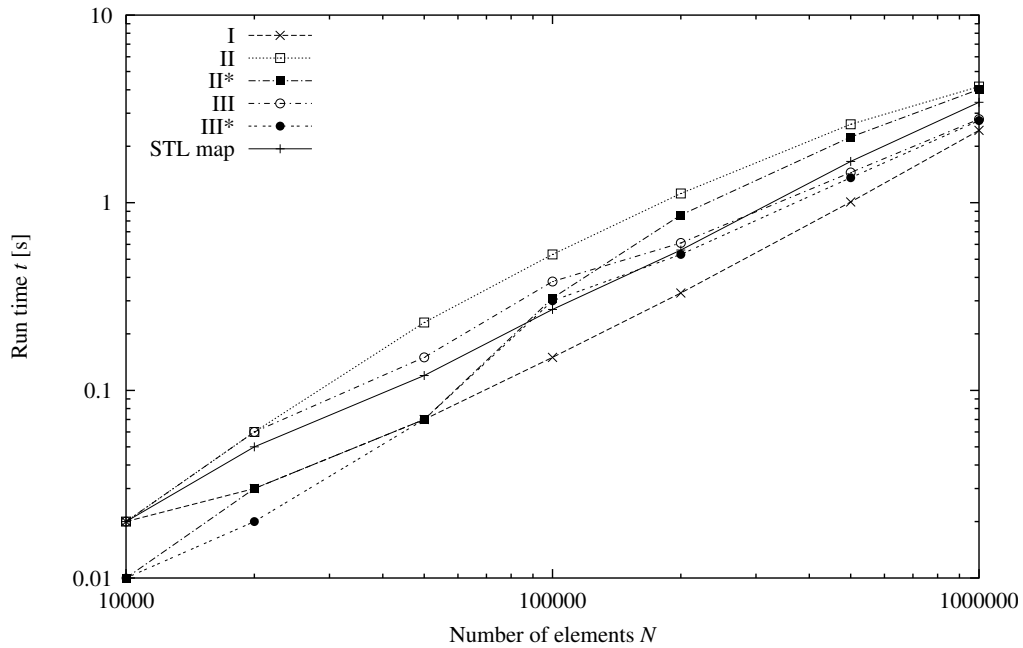
As the distribution of occupied entries in a hash table is (ideally) stochastic, a larger table will lower the probability for collisions. Resizing a partially filled table also requires rehashing.

A hash table implementation usually employs a combination of several of these strategies. For GiNaC’s hash maps, three different implementations were tested:

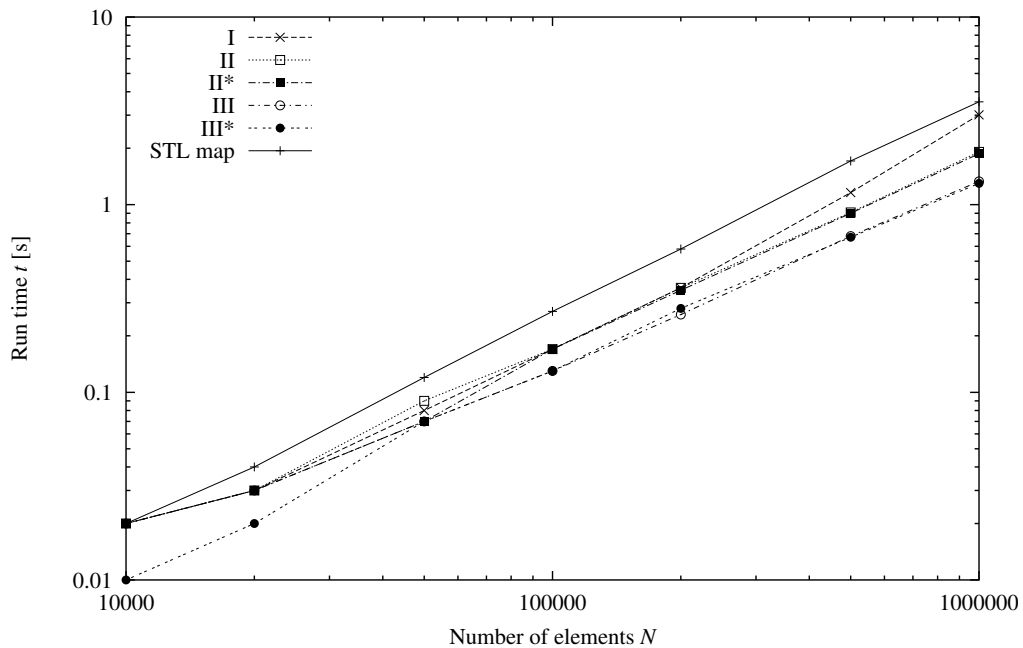
- I) Fixed table size (which has to be specified when creating the map) and chaining, with each bucket being a sorted list to minimize look-up times.
- II) Same as I, but with a hash table that grows and shrinks dynamically when the ratio of used to unused entries reaches certain thresholds. The table size is always a power of 2, and the initial size can be specified upon construction.
- III) Same as II, but with quadratic probing instead of chaining. Also, the table size in this implementation is always a prime number.

To compare these implementations, the time required to

²⁰ The C++ Standard Library as specified in [ISO 1998] does not provide hashed associative containers. They exist in some C++ implementations, including GCC, but we wanted to avoid any dependance on compiler-specific language extensions.

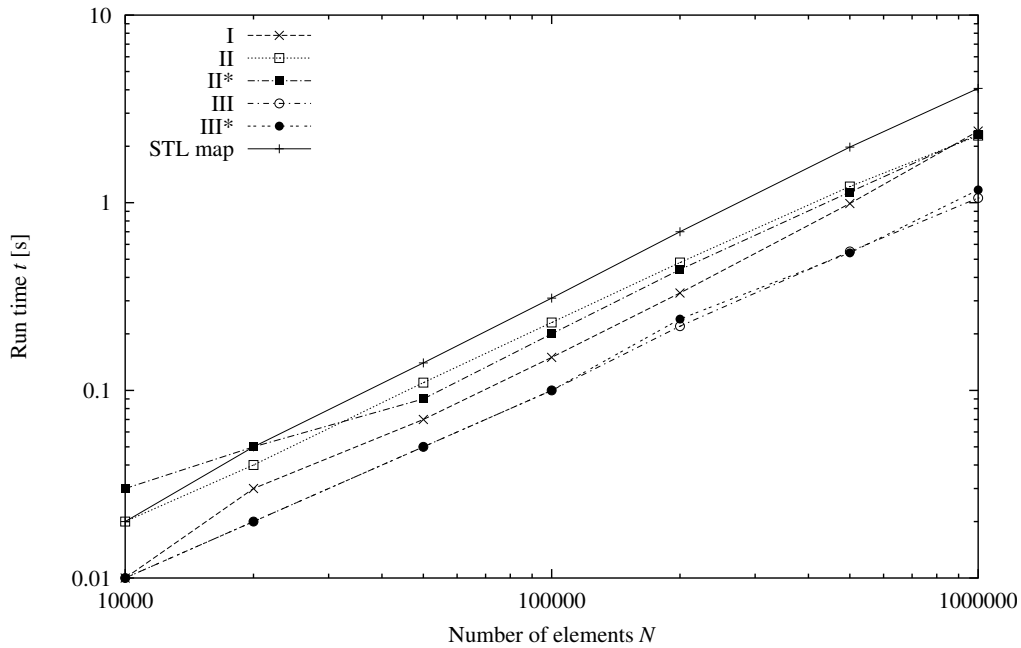


(a) Inserting elements



(b) Looking up elements

Figure 2.19: Comparison of different hash map implementations: fixed size (2^{16}) with chaining (I), dynamic size with chaining (II = initial size 32, II* = initial size 2^{16}), dynamic size with quadratic probing (III = initial size 31, III* = initial size 98317). Implementation III is used in GiNaC. For reference, the performance of the STL `map` container of GCC 3.3.2 is also included. (cont. on next page)



(c) Erasing elements

1. insert N key-value pairs into the map,
2. retrieve the values for all N keys,
3. erase all N elements

was measured for different values of N for each of the implementations and for the STL `map` container. The results are shown in fig. 2.19.

All hash map implementations have faster element look-up than the tree-based STL maps, which is of course the desired property of a hashed container. The implementations with dynamic table sizes show the expected performance hit when the number of inserted elements exceeds the initial table size (visible in fig. 2.19(a) for II* and III* at $N = 100000$). However, the look-up times (fig. 2.19(b)) which are the most important measure of performance are asymptotically better for dynamically-sized tables due to the reduction of hash collisions. Also, all three diagrams show the overhead of implementation II with respect to implementation III caused by the hash chain lists.

Based on these results, implementation III was chosen for GiNaC's `exhashmap` class template. With minor exceptions²¹ it supports the same set of operations as `map`.

²¹ `exhashmap` doesn't provide reverse iterators, the `lower_bound()` and `upper_bound()` methods, nor the `<` comparison operator between maps.

2.21 Substitutions and Pattern Matching

“Find” and “find-replace” operations are common for applications that maintain and manipulate composite object structures, whether they be word processors or symbolic calculation systems. The functionality of these operations has evolved significantly over the course of GiNaC’s history. Although they still don’t play as important a role in GiNaC as they do, for example, in FORM where substitutions constitute the basic building blocks of symbolic manipulation, GiNaC now has many features in common with it.

2.21.1 Simple Substitutions

Since its very first versions, GiNaC provided two methods for finding and replacing (substituting) elements of expressions:

- `has(s)` returns `true` if the expression it is invoked upon is equal to `s` or contains `s` as one of its subexpressions, and `false` otherwise.
- `subs(s == r)` returns a newly constructed expression in which all occurrences of `s` are replaced by `r`.²² There is another variant, `subs(ls, lr)`, that simultaneously replaces all occurrences of elements of the list `ls` with their corresponding elements in the list of equal length `lr`. Recent versions of GiNaC offer a third variant, `subs(m)`, where `m` is a mapping `ex→ex`, currently implemented as an STL `map<ex, ex, ex_is_less>`, and with the same semantics as the two-list version of `subs()`. The reason for this will be explained in section 2.21.3.

Originally, `subs()` only allowed substituting symbols and indices, albeit by arbitrary expressions. In GiNaC version 0.8.1, this was generalized to permit substituting entire subexpressions. With this, it was possible to, for instance, replace all occurrences of $\cos x$ in an expression with $\sqrt{1 - \sin^2 x}$:

```
1 symbol x("x");
2 ex e = pow(sin(x), 2) + pow(cos(x), 2);
3 e = e.subs(cos(x) == sqrt(1-pow(sin(x), 2)));
4 // e gets simplified to 1
```

However, the subexpressions have to match exactly. When invoked in this way, the substitution is only performed when the argument of the cosine function is precisely the symbol x . To replace expressions like $\cos(x-1)$ or $\cos y$, one would have to call `subs()` again with different arguments. It was not possible to replace *all* instances of the cosine function, independently from their arguments.

2.21.2 Pattern Matching

Recognizing that a more flexible substitution feature would be useful, GiNaC 0.9.0 implemented *pattern matching* to allow substituting patterns of expressions. A *pattern* is any

²² The notation with the `==` operator is a leftover from MAPLE.

symbolic expression containing one or more `wildcard` objects, with each wildcard acting as a placeholder for an arbitrary expression. This makes it possible to replace functions with arbitrary arguments:

```
1 ex e = pow(sin(x-1), 2) + pow(cos(x-1), 2);
2 e = e.subs(cos(wild()) == sqrt(1-pow(sin(wild()), 2)));
```

Wildcards can be numbered, allowing the use of more than one wildcard in a pattern, each one representing a different subexpression:

```
1 ex e = pow(x, 2) + pow(x-1, -2*y);
2 e = e.subs(pow(wild(1), wild(2)) == wild(1)*wild(2));
3 // e is now 2*x-2*(x-1)*y
```

Wildcards appearing in the replacement expression of a `subs()` call will themselves be replaced by the subexpressions matching the corresponding wildcards in the substitution pattern. It is not necessary, however, that all wildcards in the pattern also appear in the replacement expression. For example, the following code will replace all occurrences of the cosine function with 1 (which is a useful approximation if it is known that the arguments of the cosine are close to 0):

```
1 ex e = cos(x) + cos(1/y);
2 e = e.subs(cos(wild()) == 1);
3 // e is now 2
```

In addition to extending `has()` to also allow patterns as its argument, two new methods were added to GiNaC:

- `match(s)` works like `has()` but doesn't recurse into subexpressions. For example, $1 + x^2$ *has* $x^?$ (with $? = 2$) but it doesn't *match* $x^?$ (but x^2 does). Essentially, it is a version of `is_equal()` that accepts patterns. There is a second variant, `match(s, l)` that returns the list `l` of wildcard replacements necessary for the match. In the example, the returned list would contain the single element `wild() == 2`.
- `find(s, l)` appends all subexpressions matching the pattern `s` to the list `l`. It returns `true` if any matches were found, and `false` otherwise.

An application of `find()` that is used frequently in `xloops` is to assemble a list of all appearances of a certain function (or a set of functions) in an expression. The expression can then be collected in terms of these functions so each function will only appear once, making series expansion more efficient (this is, for example, used by the `Diagram1LoopNPt()` functions described in section 5.5.4).

One characteristic of the implementation of pattern matching in GiNaC is that it is still entirely *syntactic*, i. e. the object structure of the expression to be matched has to be the same as that of the pattern. For example, in the substitution

```
1 ex e = 1 + x + pow(x, 2) + pow(x, 3);
2 e = e.subs(pow(x, wild()) == pow(y, wild()));
3 // e is now 1+x+y^2+y^3
```

the term x is not substituted because, although *algebraically* equal to x^1 it doesn't match the pattern syntactically as it is not a `power` object.²³

Sums and products are an exception to the strict syntactic matching. Since both the pattern and the expression to be matched are subject to automatic and unpredictable term reordering, it was deemed impractical if, for instance, `x+y` couldn't match `x+wild()` just because the latter might be rewritten as `wild()+x` by GiNaC.

The fundamental pattern-matching method which all other methods (`has()`, `find()`, and `subs()`) build upon is `match()`, which employs a simple, recursive algorithm. Its default operation, implemented in `basic::match(s, l)`, can be outlined as follows:

1. If the pattern `s` is a single `wildcard` object, check the replacement list `l` for an occurrence of this wildcard. If one is found, its associated replacement expression must be equal to the current expression (so once a wildcard in the pattern has matched a certain subexpression, all further occurrences of the wildcard must always refer to the same subexpression). If the wildcard doesn't appear in `l`, append a new element of the form `s == *this` to `l` and return `true`.
2. If the pattern is not a single wildcard, compare the type IDs of the current expression and the pattern. If they are not equal, return `false`.
3. Next, check whether the current expression and the pattern have the same number of subexpressions (possibly zero). If they don't, return `false`.
4. If there are no subexpressions, return the result of `is_equal(s)`.
5. If there are subexpressions, check whether the attributes of the current object and the pattern object are equal. This is accomplished by calling the method `match_same_type()` which operates like `is_equal_same_type()` but doesn't compare subexpressions.
6. If the attributes are the same, check whether all of the current expression's subexpressions match the corresponding subexpression of the pattern. If this is the case, return `true`, otherwise `false`.

There are only two classes that override `basic::match()`: Firstly, the `wildcard` class itself requires matches to always be exact, i.e. `wild(1)` only matches `wild(1)` but not `wild(2)` etc. This is necessary for `subs()` to be able to re-substitute the wildcards by their matching expressions in its final step.

Secondly, `expairseq` overrides `match()` to account for the commutativity and term reordering of sums and products, as described. GiNaC allows for one "global" wildcard in a sum or product pattern, i.e. one term that consists only of a single wildcard object, that is first set aside. Next, GiNaC tries to find a matching term in the expression for every term in the pattern. If a term can't be matched, it returns `false`. Otherwise, the remaining unmatched terms are either "gobbled up" by the global wildcard if there is one, or else all terms must have been accounted for. If neither is the case, the match fails.

²³ Chris Dams has implemented some algebraic substitution capabilities for GiNaC which we will not discuss here but which are documented in section 5.4.4. of the GiNaC tutorial [BFKV 2004].

This algorithm allows, for example, to match $x+y+z+t*u$ with $t*wild(1)+x+wild(2)$:

1. The global wildcard `wild(2)` is removed from the pattern (and stored in a local variable), and both the pattern and the expression are decomposed into their individual terms. The list of terms in the pattern is $\{t*wild(1), x\}$, and the one of the expression is $\{x, y, z, t*u\}$.
2. GiNaC then looks for a term matching $t*wild(1)$, the first term of the pattern, in the expression by searching through the list of terms. The last term $t*u$ of the expression matches with $wild(1) == u$, so both terms are removed from their respective lists, which are now $\{x\}$ and $\{x, y, z\}$.
3. The same is done with the second term (x) of the pattern, which matches the first term of the expression. Again, both terms are removed, leaving $\{\}$ and $\{y, z\}$.
4. Since the pattern term list is now empty, the sum of the remaining unmatched terms ($\{y, z\}$) is assigned to the global wildcard `wild(2)`, and the match succeeds with $wild(1) == u$ and $wild(2) == y+z$.

However, the algorithm has no provisions for coping with ambiguities, and no way to backtrack its state once a wildcard-to-subexpression mapping has been established. For instance, calling

```
1 (x+y).match(wild(1)+wild(2), 1);
```

might end up with either $wild(1) == x$ and $wild(2) == y$, or with $wild(1) == y$ and $wild(2) == x$. Consequently, the result of

```
1 ((x+y)*x).match((wild(1)+wild(2))*wild(1), 1);
```

is undefined. If matching the first factor results in $wild(1) == y$ and $wild(2) == x$, the match for the second factor will fail. A more robust implementation would have to consider both possible matches of the first factor and let subsequent operations decide which is the correct one.²⁴ As this hadn't led into any problems in practice, however, the current implementation was deemed sufficient.

`find()` and `has()` simply call `match()` first before recursively invoking themselves on the object's subexpressions. The implementation of `subs()` is a little more involved. It uses postorder recursion to first substitute in any subexpressions. It then calls `match()` sequentially on each element of the list of substitutions and, if a match is successful, returns the corresponding replacement expression in which all wildcards are re-substituted with their matching subexpressions. This last step requires temporarily disabling the pattern matching mechanism to ensure proper operation. For example, consider

```
1 ex e = cos(x);
2 e = e.subs(cos(wild()) == 1-pow(wild(),2)/2);
```

²⁴ A similar problem occurs in regular expression matching when, for example, the input aab is to be matched against $a*ab$.

In the recursion step, `subs()` tries to match `x` against `cos(wild())` which is unsuccessful. Then it tries to match the whole expression `cos(x)` which yields `wild() == x`. Next it calls `subs(wild() == x)` on `1-pow(wild(),2)/2` to obtain the final replacement expression. But this substitution must only replace the one actual instance of `wild()` (to get `1-pow(x,2)/2`), not perform another pattern matching (which would yield `x`).

The `subs()` method therefore has an *options* argument by which a flag can be passed to disable the pattern matcher (`subs_options::no_pattern`). This also speeds up the operation of `subs()` when patterns are not used, as we will show in the next section.

2.21.3 Performance Issues

The performance of `subs()` can be crucial, especially when the number of subexpressions to be substituted gets large.

Fig. 2.20 compares the run times for substituting all symbols in expressions of different sizes with GiNaC versions 1.0, 1.1, and 1.2, with enabled and disabled pattern matching, for two types of expressions:

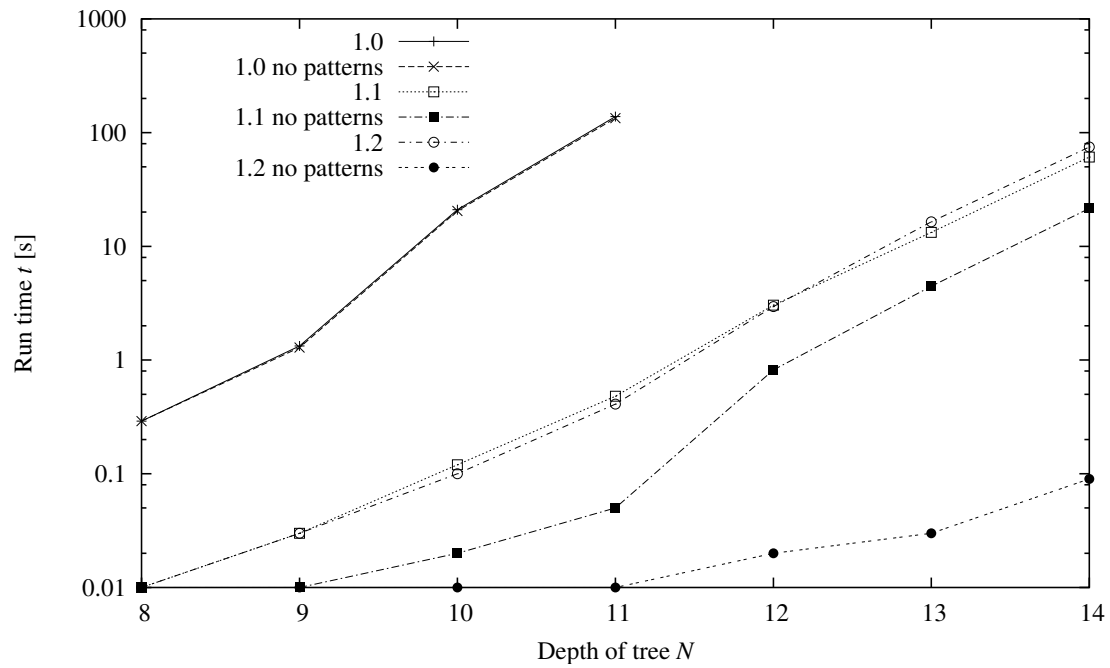
- a) Binary trees of alternating sums and products containing 2^N symbols $\{a_1, \dots, a_{2^N}\}$ on their leaf nodes.
- b) Linear expressions of the form $\sum_{i=1}^N \prod_{j=1}^i a_j$ containing N symbols $\{a_1, \dots, a_N\}$.

GiNaC 1.0 and 1.1 internally used two `lst` objects to store the substitution and replacement expressions. In GiNaC 1.0, this list was searched linearly using `op()` which, as explained in section 2.17, is an $\mathcal{O}(N^2)$ operation for lists. Changing this to use list iterators in GiNaC 1.1 resulted in a speed-up of one to two orders of magnitude. GiNaC 1.2 replaced the dual lists with a single `ex→ex` map to further reduce the look-up time which is especially apparent when no pattern matching is required. In some cases, `subs()` operations in GiNaC 1.2 are over 1000 times faster than in version 1.0. Since `subs()` is also used internally as a part of other algorithms such as rational function normalization and fraction-free matrix elimination, these too benefit from faster substitutions.

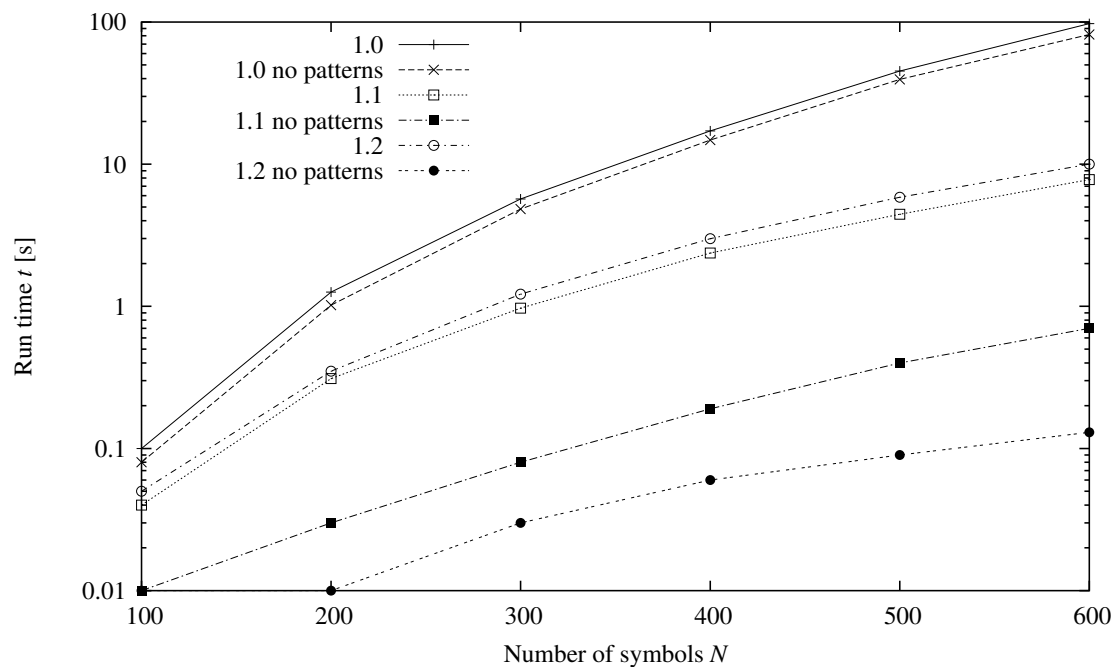
2.22 Using GiNaC in Interactive Applications

Although GiNaC was primarily designed for non-interactive use, it is occasionally useful to be able to create expressions from user input at run time. It is also convenient during development to have an interactive interface to GiNaC instead of having to go through repeated edit-compile-test cycles.

One possible approach is to use GiNaC from a C++ interpreter such as CINT [Goto 2002] which is part of the ROOT framework. This works reasonably well, but introduces some inconveniences due to restrictions in CINT and incompatibilities to the C++ standard (see [Krec 2002, section 5.2] for a detailed discussion). The GTYBALT system developed by Stefan Weinzierl [Wein 2004b] integrates GiNaC, CINT, the TEXMACS editor, and ROOT to provide an interactive interface and graphical capabilities similar to traditional computer algebra systems.



(a) Binary tree, 2^N symbols



(b) Linear polynomial, N symbols

Figure 2.20: Comparison of run times of subexpression substitutions in different GiNaC versions with two types of expressions

GiNaC itself comes with an interactive shell application called GINSH (*GiNaC interactive shell*) that emulates the behavior of interpretive-language algebra systems (i. e. symbols are used as variables which can be assigned values, etc.), and that is mainly intended for demonstration and debugging purposes. It only supports a subset of the capabilities of GiNaC (in particular, indexed expressions can't be used in GINSH) and has neither loop constructs nor conditional statements.

The central component of GINSH is an expression parser written using the scanner generator LEX and the compiler-compiler YACC [LMB 1992] that is also accessible from C++ to allow constructing GiNaC expressions from their textual representation, like this:

```
1 symbol x("x"), y("y");
2 ex e("2*sin(x+y)", lst(x, y));
```

The list of symbols that can appear in the expression has to be supplied explicitly to enable GiNaC to utilize the exact same symbols (with their unique serial numbers) in the created expression. If GiNaC created new symbols, they wouldn't be immediately accessible by the program.²⁵

Of course, the first argument to this `ex` constructor doesn't have to be a string literal. It might as well be an expression typed in by the user, for example via an HTML form. `xloops` uses the input parser to read the definitions of Feynman rules from a user-supplied model definition text file.

²⁵ The same is true for archives. Consequently, the `unarchive()` method also gets passed a symbol list as its second argument.

3 GiNaC for Physics Calculations

A novice programmer was once assigned to code a simple financial package.

The novice worked furiously for many days, but when his master reviewed the program, he discovered that it contained a screen editor, a set of generalized graphics routines, an artificial intelligence interface, but not the slightest mention of anything financial.

When the master asked about this, the novice became indignant. “Don’t be so impatient,” he said, “I’ll put in the financial stuff eventually.”

Geoffrey James, The Tao of Programming [Jame 1987]

While in the previous chapter we gave an overview of the basic design principles of GiNaC and some details of its implementation, we will now discuss some of the features that were specifically added for carrying out calculations in the Standard Model and related theories.

In particular, we will describe the role and treatment of symmetries, the handling of indexed objects and the spaces they live in, and non-commutative algebras like the Dirac algebra.

3.1 Symmetries

In general, a *symmetry* is the property of an entity to remain unchanged under a certain operation or set of operations. In mathematics and physics, these operations typically form a group.

In the terminology of GiNaC, an expression is considered symmetric when it remains algebraically equivalent under the exchange of two objects. For example, the function $f(x, y) = (x - y)^2$ is symmetric with respect to swapping its arguments: $f(x, y) = f(y, x)$.

Taking advantage of the symmetry properties of objects often leads to dramatic simplifications in a calculation. In our example, we can immediately deduce $f(x, y) - f(y, x) = 0$ from the symmetry of f , without any knowledge about the exact definition of the function. It is thus desirable that GiNaC permits defining and handling symmetries of expressions.

GiNaC is unable to recognize symmetries by itself; it relies on the user to specify them for the objects involved. Functions and objects which are predefined by GiNaC, such as the Minkowski metric tensor, already possess their correct symmetry properties, of course.

There are three basic kinds of symmetries handled by GiNaC. An expression $e[x_0, \dots, x_{n-1}]$ containing the n subexpressions $x_0 \dots x_{n-1}$ is

- *symmetric*, denoted as $+(0, \dots, n-1)$, when

$$e[x_0, \dots, x_{n-1}] = e[x_{\pi(0)}, \dots, x_{\pi(n-1)}]$$

for any permutation π .

- *antisymmetric*, denoted as $-(0, \dots, n-1)$, when

$$e[x_0, \dots, x_{n-1}] = \text{sgn } \pi \cdot e[x_{\pi(0)}, \dots, x_{\pi(n-1)}]$$

for any permutation π , where $\text{sgn } \pi$ is the sign of the permutation. This implies that

$$x_i = x_j, i \neq j \quad \Rightarrow \quad e[x_0, \dots, x_{n-1}] = 0.$$

- *cyclic-symmetric*, denoted as $@(0, \dots, n-1)$, when

$$e[x_0, \dots, x_{n-1}] = e[x_1, \dots, x_{n-1}, x_0].$$

If e does not have any of these three symmetries, we will write this as $!(0, \dots, n-1)$.

In addition to these basic types, GiNaC allows specifying more complex, mixed symmetries by arranging these types into a *symmetry tree*. For example, the property of the Riemann curvature tensor

$$R_{\alpha\beta\mu\nu} = -R_{\beta\alpha\mu\nu} = -R_{\alpha\beta\nu\mu} = R_{\mu\nu\alpha\beta}$$

of being antisymmetric in the first and second pair of indices, and symmetric under exchange of the two pairs, would be denoted as

$$+(-(0, 1), -(2, 3)).$$

Symmetry trees can be arbitrary, as long as they adhere to the following two constraints:

1. No subexpression $x_0 \dots x_{n-1}$ may be referenced more than once.
2. Each child of a $+$, $-$ or $@$ node must reference the same number of subexpressions.

These constraints remove the major source of ambiguities and drastically simplify the sorting algorithm described below.

Two classes of algebraic objects in GiNaC contain symmetry attributes:

- Functions can have a symmetry with respect to permutations of their arguments. In this case, the expression e is the function and the subexpressions $x_0 \dots x_{n-1}$ are the arguments of the function.¹
- Indexed objects can have a symmetry with respect to permutations of their indices. In this case, the expression e is the indexed object and the subexpressions $x_0 \dots x_{n-1}$ are the individual indices.

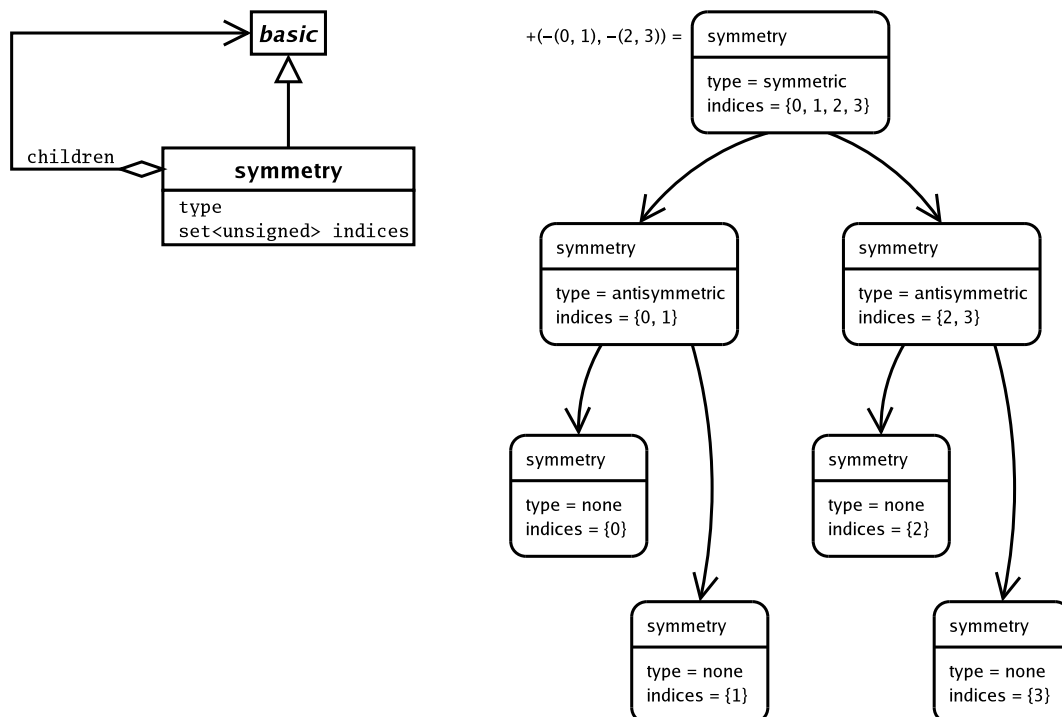


Figure 3.1: Class structure (left) and object structure (right) of symmetry trees. The indices set of each non-leaf node is the set union of the indices of the node's children.

Internally, GiNaC represents symmetry specifications as a tree of `symmetry` objects (see fig. 3.1). Although these objects don't normally appear as part of expressions, the `symmetry` class is derived from `basic` to take advantage of the automatic memory management and convenient composition facilities of algebraic objects.

Each `symmetry` object contains a `type` attribute that specifies which of the four kinds of symmetries the object represents (`symmetric`, `antisymmetric`, `cyclic`, and `none`, corresponding to $+$, $-$, $@$, and $!$, respectively), and the set of the indices² of all subexpressions referenced by the object, including all its children. The leaf nodes of a symmetry tree consist of objects with `type = none` and exactly one index.

The construction of symmetry trees is supported by the four helper functions `sy_symm()`, `sy_anti()`, `sy_cycl()`, and `sy_none()` that create and connect the nodes, and also validate the index constraints. For example, an expression representing the Riemann curvature tensor $R_{\alpha\beta\mu\nu}$ could be written like this (omitting the definition of the indices):

```

1 ex e = indexed(symbol("R"), sy_symm(sy_anti(0, 1), sy_anti(2, 3)),
2           alpha, beta, mu, nu);

```

If a symmetry has been specified for a function or indexed object, GiNaC will automati-

¹ Functions can of course have other types of symmetries, like $\sin(-x) = -\sin x$, but these have to be implemented in the numeric or symbolic evaluation routines that are specific to that function. There is no general mechanism in GiNaC that handles such symmetries.

² The use of the term *index* may be confusing. Here, it refers to the index numbers of the subexpressions $x_0 \dots x_{n-1}$. These subexpressions may or may not be actual index (`idx`) objects.

cally reorder its arguments or indices into a canonical order as a part of the anonymous evaluation of expressions (see section 2.8). This reordering yields some immediate simplifications like

$$\begin{aligned}f(x, y) - f(y, x) &= f(x, y) - f(x, y) = 0, \\g_{\mu\nu} + g_{\nu\mu} &= 2g_{\mu\nu}, \\ \epsilon_{\mu_1\mu_2\mu_3\mu_4} + \epsilon_{\mu_3\mu_2\mu_1\mu_4} &= \epsilon_{\mu_1\mu_2\mu_3\mu_4} - \epsilon_{\mu_1\mu_2\mu_3\mu_4} = 0, \\ \epsilon_{ijj} &= 0,\end{aligned}$$

as long as they follow from *local* transformations, i. e. transformations of single objects. At this stage, GiNaC doesn't yet consider symmetries involving two or more objects.

The key operation of the reordering process is the function

```
int canonicalize(vector<ex>::iterator v, symmetry s)
```

which employs the *Interpreter* pattern [GHJV 1995] to sort the elements of an expression vector, designated by an iterator `v` pointing to its first element, according to the rules specified by the symmetry tree `s`. The return value of this function is the overall sign introduced by the permutation, or zero if an antisymmetric object is found to have two identical subexpressions.

The operation of `canonicalize()` can be outlined as follows:

```
1 int canonicalize(vector<ex>::iterator v, symmetry s)
2 {
3     if (s has less than two indices)
4         return 1;
5
6     int sign = 1;
7     forall children c of s {
8         sign *= canonicalize(v, c);
9     }
10
11     switch (s.type) {
12     case symmetric:
13         sort elements of v referenced by s.indices;
14         break;
15     case antisymmetric:
16         sort elements of v referenced by s.indices;
17         sign *= permutation sign of the sort;
18         break;
19     case cyclic:
20         determine the smallest element of v referenced by s.indices;
21         rotate this element to the front;
22         break;
23     }
24
25     return sign;
26 }
```

After first recursively descending into the symmetry tree, multiplying all the signs accumulated in the application of the subsymmetries, `canonicalize()` reorders a subset of the elements of the expression vector using one of three sorting algorithms, according to the type of the current symmetry node.

These algorithms are implemented as generic functions, parametrized by both a *comparison* functor and a *swapping* functor. The comparison functor performs the resolution of the subexpression indirection in the `indices` attribute of the `symmetry` objects (i. e. it extracts the subexpressions from the expression vector and compares them), while the swapping functor exchanges the vector elements referred to by two symmetry nodes. Unlike conventional sorting algorithms, the functors allow treating sets of elements as if they were single elements. For example, in the final step of canonicalizing the indices of a Riemann tensor $R_{\alpha\beta\mu\nu}$, GiNaC must compare, and eventually swap, the two *blocks* of indices $\alpha\beta$ and $\mu\nu$.

Since all of the indirection resolution is encapsulated in the functors, the sorting functions are passed the range of symmetry objects that apply, instead of the range of vector elements to be sorted:

```
1 sort(s.children.begin(), s.children.end(), sy_is_less(v), sy_swap(v));
```

Effectively, `canonicalize()` is a recursive sorting function controlled by the “symmetry language” defined by the recursive composition of `symmetry` objects.

Reordering function arguments and indices is not the only way in which GiNaC exploits symmetries. Other uses such as calculating contractions between symmetric tensors will appear in the following sections.

3.2 Special Algebras

The fundamental algebra implemented by GiNaC is that of the field of complex numbers. Symbols in GiNaC represent complex quantities by default, and the operators `+` and `*` perform the operations which define this field.

In physics, and especially in high-energy physics, there is frequently a need to calculate with elements of different algebras. In particular, the calculation of Feynman amplitudes in the Standard Model as implemented by `xloops` requires

- matrices,
- Lorentz vectors and tensors,
- the algebra of Dirac matrices,
- algebras of objects related to symmetry groups, such as the structure constants and generators of the Lie algebra of $SU(N)$.

An implementation of Lorentz vectors and the $SU(3)$ color algebra has already been developed in [Frin 2000, section 2.4]. For this thesis, the implementation was generalized by allowing more universal types of indexed expressions and indices, and by moving calculation and transformation rules out of the specific Lorentz and color classes into common, reusable base classes and functions. Furthermore, higher-order tensors and the Dirac algebra have been added, and the matrix class implemented in [Krec 2002, section 4.7] has been extended to integrate seamlessly with the index mechanism described in the next section.

3.2.1 Indexed Expressions

All of the algebras listed above commonly involve the use of indices. From GiNaC’s point of view, an *index* is at first simply a subscript or superscript to an expression.³ Semantically, the presence of an index signifies that the indexed expression represents a *collection* of objects, selected by individual values of the index, but treated as a single entity, without the need to list the components of that collection individually.

For example, an element \mathbf{x} of a three-dimensional vector space can, after choosing a coordinate system, be represented by three components x_1 , x_2 , and x_3 , which in turn can be combined to x_i with the index i running over the *index set* $\{1, 2, 3\}$. The expression x_i now stands for the individual components of \mathbf{x} (for a specific value of i) as well as \mathbf{x} itself, if i is interpreted as an indeterminate.

Modern mathematics favors the index-free notation \mathbf{x} because it allows talking about the properties of elements of vector spaces (and other classes of objects, in general) without resorting to a specific representation for them. But in physics, indices are often written down explicitly because concrete calculations typically require working with specific representations anyway. Feynman rules in particular are usually given with indices because their presence simplifies the step of assembling a Feynman amplitude to a plain multiplication of individual factors, mitigating the need for additional calculation prescriptions.⁴

For this reason, GiNaC supports the use of explicit indices. Any algebraic expression e can be augmented by an arbitrary number of indices $i_1 \dots i_n$ to yield the indexed expression $e_{i_1 \dots i_n}$. Although `xloops` only uses indices which are simple symbols or integer numbers, such as in p^μ or g_{11} , the *value* of an index in GiNaC can itself be an arbitrary expression. This allows the user to work with more complex indices like $2n - 1$, if desired.

In addition to its value, an index in GiNaC carries a *dimension*, which for a finite index set I_k , $i_k \in I_k$ is the cardinality of the set, $\dim(i_k) := |I_k|$. A physical four-momentum p^μ , for example, has a four-dimensional index μ , $\dim(\mu) = 4$. In general, the index dimension specifies the dimension of the (sub)space the indexed object “lives” in with respect to the index, and to some extent characterizes that space. Since the technique of dimensional regularization generalizes the four-dimensional loop momentum space to a Minkowski space with unspecified dimensionality D , and the parallel/orthogonal decomposition procedure described in section 5.2 requires working with D_{\parallel} - and $(D - D_{\parallel})$ -dimensional subspaces of this momentum space (D_{\parallel} being the number of independent external momenta), index dimensions in GiNaC can also be arbitrary expressions. Because the dimension may appear in the result of a calculation with indexed expressions (typically when computing traces), and because an object may have more than one different type of index (such as Lorentz and spinor indices) with different dimensions, the dimension is stored as a part of the index.

Fig. 3.2 shows the fundamental classes for handling indices in GiNaC. The `indexed` class, derived from `exprseq` which stores a vector `seq[]` of subexpressions, represents an object carrying indices. The *base expression*, i. e. the expression being indexed, is stored in the

³ Exponents such as the 2 in x^2 are not indices in GiNaC, although they are also written as superscripts.

⁴ As we shall see in section 3.4.1, however, GiNaC makes some compromises here, for example by omitting spinor indices and requiring explicit traces of Dirac strings.

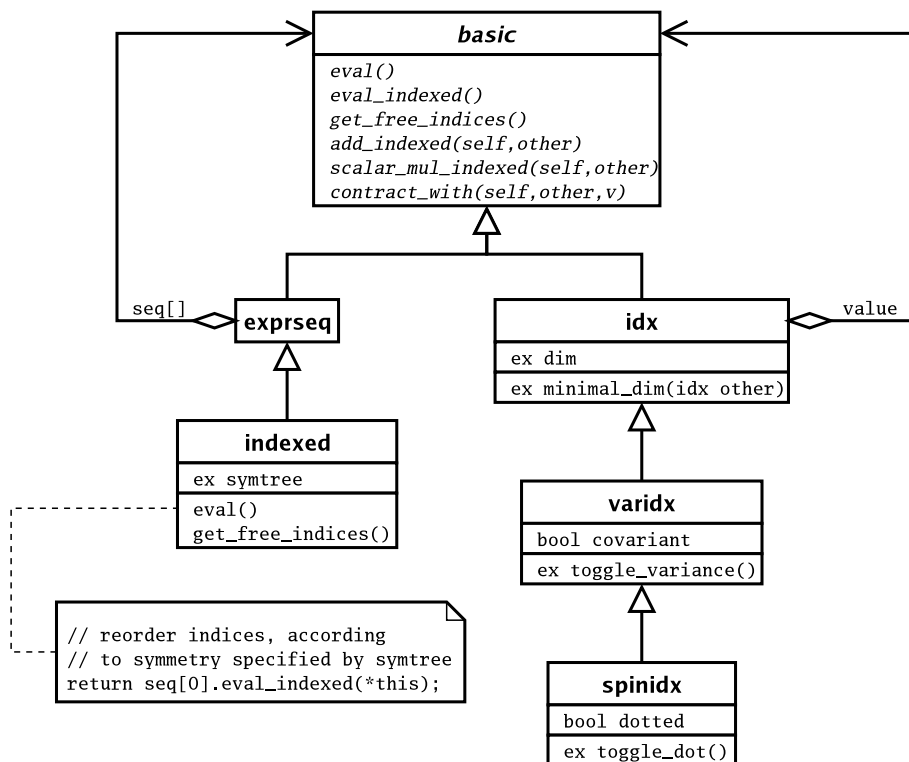


Figure 3.2: Class structure of indexed objects and indices, and the member functions that play a role in the simplification of indexed expressions.

first element `seq[0]` of the vector, while the subsequent elements `seq[1]...seq[n]` hold the indices. It is possible for an `indexed` object to have no indices, in which case the vector only contains the base expression. Such degenerate indexed expressions are used in the implementation of the Dirac algebra (see section 3.4.1 for details).

Indices must belong to the class `idx`⁵ or a subclass, which is enforced by the constructors of `indexed`. `idx` stores the index value as a subexpression, but the dimension, though also of type `ex`, as an attribute.

Subclasses of `idx` add further attributes. The `varidx` class stores a flag which marks the index as either co- or contravariant, which makes it print itself as a sub- or a superscript, respectively (indices without variance are always printed as subscripts). The method `varidx::toggle_variance()` returns a new index with the same value but opposite variance. Indices of class `varidx` are used in `xloops` for Lorentz indices.

The class `spinidx` represents spinor indices in the Weyl-van-der-Waerden formalism [Ditt 1998]. In addition to being co- or contravariant, a `spinidx` can be dotted or undotted, with the dot representing complex conjugation of the associated spinor.⁶

Fig. 3.3 illustrates the object structure of an indexed expression considering as example the four-momentum p^μ .

⁵ The name `idx` was chosen to avoid a clash with the the BSD C library function `index()` in an early version of GiNaC, before the proper use of namespaces.

⁶ The `spinidx` class is not currently used in `xloops`.

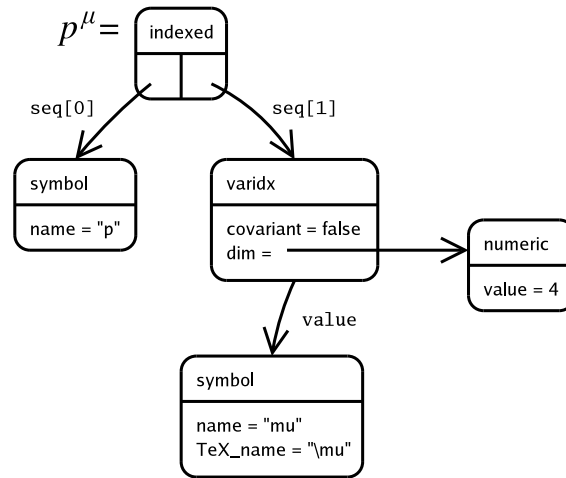


Figure 3.3: Possible representation of the four-momentum p^μ in GiNaC

3.2.2 Predefined Tensors

Some frequently-used tensors are built into the GiNaC library. These are all derived from the common abstract base class `tensor`:⁷

- `tensdelta`

The Kronecker delta δ_{ij} representing the identity matrix, defined by

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

- `tensmetric`

A general (symmetric and regular) metric tensor g_{ij} without any specific matrix representation.

- `minkmetric` (derived from `tensmetric`)

The Minkowski metric tensor $\eta^{\mu\nu}$, with user-selectable, positive or negative signature, the latter being the default. Its matrix representation is $\text{diag}(-1, 1, 1, \dots)$ or $\text{diag}(1, -1, -1, \dots)$, respectively.

- `spinmetric` (derived from `tensmetric`)

The antisymmetric Weyl spinor metric ϵ_{AB} defined by the matrix representation

$$\epsilon_{AB} = \epsilon_{\dot{A}\dot{B}} = \epsilon^{AB} = \epsilon^{\dot{A}\dot{B}} = i\sigma^2 = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}.$$

This is used in conjunction with the `spinidx` class.

⁷ The term *tensor* is used loosely here. Objects of class `tensor` are not required to obey tensor transformation rules.

- `tensepsilon`

The totally antisymmetric Levi-Civita tensor $\epsilon_{i_1\dots i_n}$, where the number n of indices is equal to the dimension of the space. In a four-dimensional Minkowski space, $\epsilon^{0123} = -\epsilon_{0123} = 1$.

Instances of these classes are always used as base expressions of indexed objects. For the most common uses of these tensors, utility functions are provided that construct a proper composite indexed expression. For example, the function `lorentz_g(i1, i2)` constructs and returns an `indexed` object with a `minkmetric` object as its base, the supplied `varidx` indices `i1` and `i2`, and the symmetry $+(0,1)$. To keep the total number of objects in expressions involving many metric tensors small, both the `minkmetric` object and the symmetry tree are created as flyweights and reused in subsequent calls of `lorentz_g()`.

Reviewing the GiNaC class hierarchy displayed in fig. 2.4, the reader will notice that there are many more classes derived from `tensor`. These belong to the implementation of the Dirac and $SU(3)$ algebras and will be described in sections 3.4.1 and 3.4.2, respectively.

Furthermore, `matrix` objects can be used with indices to allow writing matrix expressions in tensor notation (with explicit indices), and to allow substituting symbolic vectors and second-rank tensors with their matrix representations, if desired.

There are many calculation and simplification rules built into the predefined tensors, such as

$$\begin{aligned}\eta^{\mu\nu} - \eta^{\nu\mu} &= 0, \\ g^{\mu\nu} p_\nu &= p^\mu, \\ g^\mu{}_\mu &= \delta^\mu{}_\mu = \dim(\mu), \\ \epsilon_{\mu\nu\rho\sigma} p^\mu p^\nu &= 0.\end{aligned}$$

See appendix B.1 for a complete list. The first rule is a direct result of the index symmetry of the η tensor, as explained in section 3.1. In the next section we will discuss how the other rules are implemented.

3.3 Simplifying Indexed Expressions

Expressions containing indexed objects can often be brought into a more compact form by applying certain transformation rules and exploiting symmetry properties, especially when these expressions are obtained by assembling generic factors, as is the case when constructing the Feynman amplitude for a diagram from general Feynman rules for vertices and propagators.

In GiNaC, this is carried out in two stages. The automatic expression evaluation method `eval()` (see section 2.8) performs operations on individual indexed objects that are at most of complexity $\mathcal{O}(N \log N)$, with N being the number of indices of the object:

- Indexed objects whose base expression is zero evaluate to zero. This allows simplifications like $(p - q)^\mu|_{p=q} = 0$.

- Numeric factors in the base expression are pulled out, so e. g. $(p + q)^\mu|_{p=q} = (2p)^\mu = 2p^\mu$.
- The indices are sorted according to the symmetry specified for the object, as described in section 3.1.
- Matrices and built-in tensors perform traces, contractions and (for tensors with matrix representations) evaluate to individual elements when all indices are numeric. Examples for these transformations are

$$\begin{aligned}\delta_{ii} &= \text{dim}(i), \\ \epsilon^\mu{}_{\mu\rho\sigma} &= 0, \\ \eta_{22} &= -1.\end{aligned}$$

Transformations of higher complexity that typically involve more than one indexed object are not carried out automatically but have to be initiated by the user, by invoking the method `simplify_indexed()`. These transformations include:

- Evaluating contractions like $\delta_{ij}X_j = X_i$ and $\gamma^\mu\gamma^\alpha\gamma_\mu = (2 - \text{dim}(\mu))\gamma^\alpha$.
- Evaluating sums, contractions (products), and multiplications with scalars of indexed matrices.
- Detecting contractions that vanish for symmetry reasons (contracting an antisymmetric expression with a symmetric one), such as $g_{\mu\nu}(p^\mu q^\nu - p^\nu q^\mu) = 0$.
- Canonicalizing summation indices, for example $x_i y_i + x_j y_j = 2x_i y_i$.
- Recognizing scalar products and optionally replacing them by user-supplied values. For example, if p and q are known to be perpendicular, then GiNaC can substitute $p^\mu q_\mu$ with 0.

The implementation of these transformations in GiNaC is designed to encapsulate tensor-specific behavior in member functions of the respective tensor classes, but to provide general algorithms such as index canonicalization in base classes like `indexed`. In this way, new tensors with their own algebraic rules can be added without requiring changes to existing tensor classes, but also without having to duplicate rules that are valid for any calculation with indexed objects.

Most of the transformations performed by `simplify_indexed()` involve summations over index sets, which leads to the concept of dummy indices.

3.3.1 Dummy Indices

To all indexed expressions GiNaC applies the Einstein summation convention, i. e. repeated indices imply a sum over all elements of the index set:

$$X_i Y_i \equiv \sum_{i \in I} X_i Y_i.$$

Indices with variance are only summed over when they are of opposite variance:⁸

$$X_\mu Y^\mu \equiv \sum_\mu X_\mu Y^\mu, \text{ but } X_\mu Y_\mu \not\equiv \sum_\mu X_\mu Y_\mu.$$

Following the parlance of [Capr 1999], we will refer to summation indices as *dummy indices*, and to indices that are not summed over as *free indices*.

An expression can have both free and dummy indices. For example, in

$$\epsilon_{\mu\nu\rho\sigma} p^\mu q^\nu$$

the indices $\{\mu, \nu\}$ are dummy indices while $\{\rho, \sigma\}$ are free indices. The method `ex::get_free_indices()` returns the set of free indices of an arbitrary expression and also checks the consistency of the indices of terms in sums. Adding two expressions with different sets of free indices, such as $p^\mu + q^\nu$, is invalid and reported as an error.

GiNaC only considers pairs of dummy indices. Summing indices that are repeated more than once, as in $X_i Y_i Z_i$, is not properly supported. Also, both indices must be `idx` objects. Expressions like the general form of a polynomial $P(x) = a_n x^n$ where the n in x^n is an exponent can be created but are not treated as implicit sums.

Two indices form a dummy index pair when

1. They are of the same class.
2. They have the same value.
3. That value is a single symbol. This naturally excludes numeric indices like $p_0 q_0$, but also indices that are composite expressions like $a_{2n+1} b_{2n+1}$ because there is no trivial way for GiNaC to deduce the effective summation range of such indices.
4. They have opposite variance if they are of class `varidx`.
5. Their dimensions are comparable.

This last point plays a role when considering contractions between tensors in different subspaces. For example, contracting a D -dimensional loop momentum l^μ with a four-dimensional external momentum p_μ results in a *four*-dimensional scalar product

$$l^{\mu[D]} p_{\mu[4]} = \sum_\mu^4 l^\mu p_\mu = l^{\mu[4]} p_{\mu[4]}$$

(the values in square brackets denote the index dimensions). In general, a contraction of two indices i_1, i_2 with different dimensions $\dim(i_1) \neq \dim(i_2)$ is treated as a summation over the minimum of dimensions $\min(\dim(i_1), \dim(i_2))$.⁹ GiNaC thus effectively projects

⁸ This behavior can be used to turn off the summation for indices without variance, by defining them as variant indices and giving them all the same variance.

⁹ This is of course not correct in the general case. The summation should run over the intersection of the index sets of i_1 and i_2 , but since GiNaC has no way of specifying explicit index sets we assume that one of the sets is a subset of the other, which is always true for the indices used by `xloops`.

the higher-dimensional object down to the lower-dimensional space.¹⁰ For instance, multiplying a D -dimensional vector p^μ with a four-dimensional metric tensor yields a four-dimensional covector:

$$g_{\mu[4]\nu[4]}p^{\nu[D]} = p_{\mu[4]}.$$

GiNaC therefore needs a way to determine the minimum of the dimensions of any two indices. While this is obvious for numeric dimensions, it could potentially get quite involved for an arbitrary number of symbolic dimensions, requiring the user to explicitly define the nesting of subspaces ($A < B < C$). Since, however, calculations in dimensional regularization only involve one symbolic dimension D for an arbitrary number of loop momenta, GiNaC takes a shortcut here by treating symbolic dimensions as having a positive value larger than any numeric dimension. This yields the correct ordering of dimensions for `xloops` without having to give details about D . The method `idx::minimal_dim(idx other)` returns the minimum of the dimensions of the invoked-on index and the passed second one.

3.3.2 Automatic Evaluation

As described in section 2.8, the anonymous evaluator `eval()` only performs local transformations on individual objects. This constitutes a problem for indexed expressions, since, for example in the case of the trace of the delta tensor

$$\delta_{ii} = \dim(i),$$

the “knowledge” about this rule should reside inside the `tensdelta` class (as it is specific to the delta tensor), but the `tensdelta` instance is wrapped inside the outer `indexed` object, and so `tensdelta::eval()` has no access to the indices.

The solution is to have `indexed::eval()` invoke another method, `eval_indexed(i)`, on its base expression after performing index canonicalizations, passing a reference to itself as an argument (see fig. 3.2). `tensdelta::eval_indexed()` can then examine the indices and act accordingly. All predefined tensors and the `matrix` class implement `eval_indexed()` to compute traces and contractions, and to provide access to individual tensor or matrix components when all of their indices are numeric.

3.3.3 Manual Evaluation

As mentioned, complex transformations of an expression containing indices are carried out by the `simplify_indexed()` method which first expands the expression to ensure that all products don’t have any sums as factors, and then works through the expression in a recursive fashion.

Two key operations performed by `simplify_indexed()` are the *renaming* and *repositioning* of dummy indices. Since the names of summation indices can be changed arbitrarily, canonicalizing their names allows immediate simplifications based on purely syntactic comparisons.

¹⁰ Alternatively, this can be viewed as embedding the lower-dimensional object in the higher-dimensional space, and treating the extra components as zero.

Consider, for example, the expression

$$X^{\mu\nu}Y_{\mu\nu} - X^{\mu\rho}Y_{\mu\rho}$$

which, by substituting $\rho \rightarrow \nu$ in the second term, is obviously equivalent to

$$X^{\mu\nu}Y_{\mu\nu} - X^{\mu\nu}Y_{\mu\nu} = 0.$$

To achieve this equivalence, GiNaC keeps track of the set G (the “global” set) of previously encountered dummy indices and compares it with the set L_i (the “local” set) of dummy indices of each subsequent term i . If $|L_i| > |G|$, the extra indices in L_i are added to G . GiNaC then computes the two sets $U_L := L_i \setminus G$ of unique local and $U_G := G \setminus L_i$ of unique global indices, and renames all indices in U_L to the corresponding indices in U_G .

In the example, starting with $G = \{\}$ we get $L_0 = \{\mu, \nu\}$ for the first term. $|G| = 0$, so we add all indices from L_0 to G to get $G = \{\mu, \nu\}$. Since G and L_0 are now identical, $U_L = U_G = \{\}$, and no renaming takes place. The second term has the local set $L_1 = \{\mu, \rho\}$ and because $|L_1| = |G|$, no indices are added to G . The set differences yield $U_L = \{\rho\}$ and $U_G = \{\nu\}$, hence we rename $\rho \rightarrow \nu$ in the second term. Subsequent automatic evaluations in the `add` class will then yield the final result 0.

Dummy index repositioning takes advantage of the fact that

$$p^\mu q_\mu = g_{\mu\rho} p^\mu q^\rho = g_{\rho\mu} p^\mu q^\rho = p_\rho q^\rho = p_\mu q^\mu$$

for any symmetric metric g , to canonicalize the variance of dummy index pairs and obtain simplifications like

$$p^\mu q_\mu + p_\mu q^\mu = p^\mu q_\mu + p^\mu q_\mu = 2p^\mu q_\mu$$

by placing the contravariant index in a pair first.¹¹

The main operation of `simplify_indexed()` on products of indexed objects is to evaluate contractions. Both commutative and non-commutative products are first “flattened” to a single vector of factors (because non-commutative products are grouped by the types of the objects being multiplied, see section 3.4). Then, GiNaC looks for all possible pairs of factors that are contracted with each other (i.e. those which share a dummy index pair) and invokes the method `contract_with(it1, it2, v)` on the base expression of the first object. This method takes two mutable iterators `it1` and `it2` pointing to the first and second indexed expression and a reference to the entire vector of factors `v`, and returns a `bool` value indicating whether the contraction could be successfully evaluated. If it returns `false`, GiNaC calls `contract_with()` again, this time on the base expression of the second contracted object. This allows contractions between different classes of objects to be implemented in one class only, without requiring knowledge of the other class.

For example, given the product

$$p^\mu \eta_{\mu\nu}$$

`simplify_indexed()` first invokes `contract_with()` on `p`, a symbol. The `symbol` class doesn’t know anything about indices, so `symbol::contract_with()` falls back to `basic::contract_with()` which does nothing and returns `false`. GiNaC then invokes

¹¹ This transformation is not performed for `spinidx` indices as the spinor metric is antisymmetric.

`contract_with()` on η , which is a `minkmetric` object. `minkmetric::contract_with()` evaluates the contraction by replacing the dummy index μ in p^μ with its free index ν (taking care of inserting the proper index dimension) and replacing its own indexed object $\eta_{\mu\nu}$ by 1, yielding

$$p_\nu \cdot 1$$

which evaluates to p_ν in `mul::eval()`.

Each predefined tensor class overrides `contract_with()` to implement tensor-specific behavior. The `matrix` class performs matrix and vector multiplications for indexed matrices. It is also the only class that implements the two additional methods `add_indexed()` and `scalar_mul_indexed()` which are called by `simplify_indexed()` to evaluate sums of indexed objects and products of indexed objects with scalars, respectively. This allows evaluating expressions containing indexed matrices with `simplify_indexed()` in a way similar to the evaluation of non-indexed matrix expressions with `evalm()`, as described in section 2.19.

In addition to the contraction algorithm just detailed, `simplify_indexed()` provides another mechanism that allows replacing scalar products of indexed objects (i.e. contractions with no free indices) with user-supplied values without requiring the involved objects to override `contract_with()`, but by looking them up in a table passed to `simplify_indexed()`. This is particularly useful when ordinary symbols are used to denote vectors, as in

$$(p^\mu + q^\mu)(l_\mu + q_\mu).$$

If it is known that q is light-like and that the product of p and l is $p_0 l_0$, the user can pass the set of scalar products

$$\begin{aligned} q \cdot q &\rightarrow 0 \\ p \cdot l &\rightarrow p_0 l_0 \end{aligned}$$

to `simplify_indexed()` to obtain the result

$$p_0 l_0 + q^\mu l_\mu + p^\mu q_\mu.$$

The scalar product definitions are stored in an object of class `scalar_products` as a mapping $(e_1, e_2, D) \mapsto e_3$ (symmetric in e_1 and e_2), where D is the (possibly unspecified) effective dimension of the contraction.

The final task of `simplify_indexed()` is to detect terms that vanish or cancel each other out due to symmetry reasons. In this step, GiNaC only considers symmetries with respect to dummy indices. By symmetrizing all terms containing contractions over the involved dummy indices and combining the resulting terms, most cancellations in contractions can be found easily, since any indexed expression is supposed to be symmetric under exchange of its dummy indices.

For example, in

$$\epsilon_{\mu\nu\rho\sigma}(p^\mu q^\nu + p^\nu q^\mu)$$

`simplify_indexed()` first expands the product to get

$$\epsilon_{\mu\nu\rho\sigma} p^\mu q^\nu + \epsilon_{\mu\nu\rho\sigma} p^\nu q^\mu.$$

Both terms are then symmetrized over $\{\mu, \nu\}$, yielding

$$\begin{aligned} & \frac{1}{2}(\epsilon_{\mu\nu\rho\sigma}p^\mu q^\nu + \epsilon_{\nu\mu\rho\sigma}p^\nu q^\mu) + \frac{1}{2}(\epsilon_{\mu\nu\rho\sigma}p^\nu q^\mu + \epsilon_{\nu\mu\rho\sigma}p^\mu q^\nu) \\ &= \frac{1}{2}(\epsilon_{\mu\nu\rho\sigma}p^\mu q^\nu - \epsilon_{\mu\nu\rho\sigma}p^\nu q^\mu) + \frac{1}{2}(\epsilon_{\mu\nu\rho\sigma}p^\nu q^\mu - \epsilon_{\mu\nu\rho\sigma}p^\mu q^\nu) \end{aligned}$$

by canonicalizing the indices of the ϵ tensor and using its antisymmetry. Re-expanding the expression then gives the result 0.

Other kinds of symmetries such as Jacobi or Bianchi identities are not handled by `simplify_indexed()`, but those do not appear in the calculations performed by `xloops`.

3.4 Non-Commutative Algebras

There are three classes of objects with non-commutative products built into GiNaC 1.2:

- matrices (class `matrix`),
- elements of the Dirac algebra γ^μ and related items like γ^5 and slashed momenta $\not{p} \equiv \gamma^\mu p_\mu$ (class `clifford`),
- the generators T_a of the Lie algebra of $SU(3)$ (class `color`); other objects associated with this algebra, like the structure constants f_{abc} , are commutative.

Most of the basic classes and algorithms for dealing with non-commutative products in GiNaC have already been described in [Frin 2000] and [Krec 2002] and shall only be briefly summarized here (see also fig. 3.4).

Non-commutativity in GiNaC is a property of the algebraic objects. The `*` operator figures out by itself whether its factors commute and either creates a `mul` container for a commutative product or an `ncmul` container for a non-commutative one. The `ncmul` class stores its factors in a simple expression vector (unlike the pair-wise representation employed by `mul` which we described in section 2.6.4) and also doesn't automatically canonicalize the order of the factors.

The commutation properties of an algebraic class are implemented in the two methods `return_type()` and `return_type_tinfo()`. `return_type()` returns one of three values:

- `commutative`
An object that commutes with any other object.
- `noncommutative`
A non-commutative object of a “pure” type.
- `noncommutative_composite`
A non-commutative object of composite type.

The return value of `return_type_tinfo()` is an ID that determines which “type” a non-commutative object belongs to. For simplicity, the `matrix`, `clifford`, and `color` classes

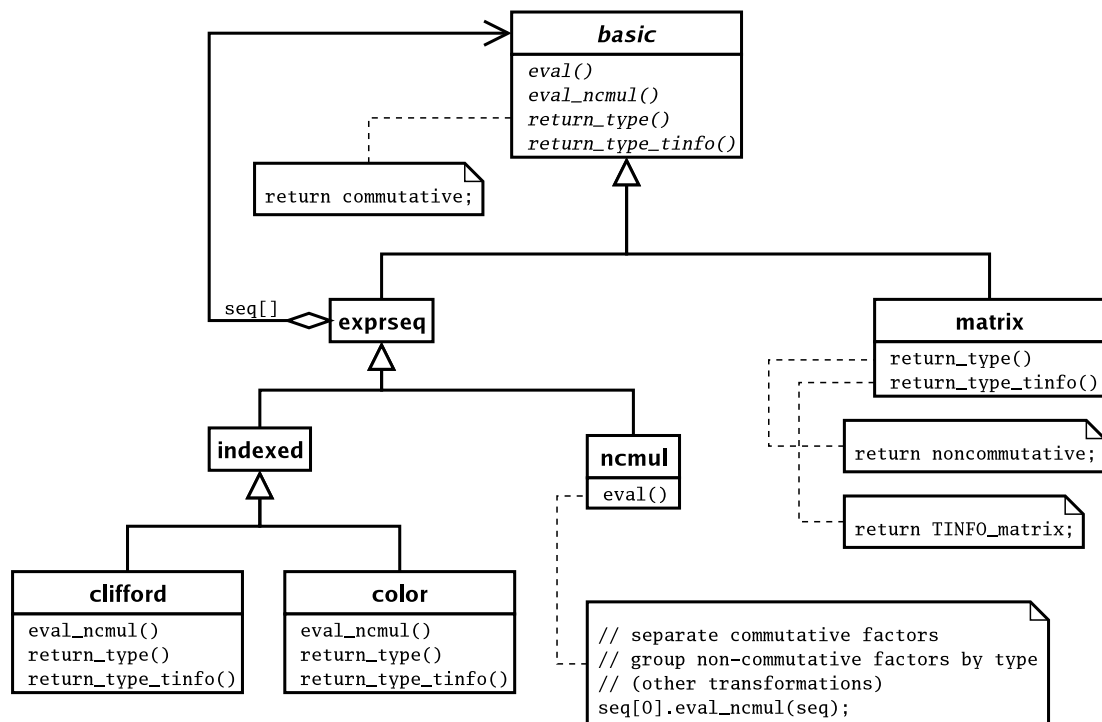


Figure 3.4: Classes and methods for non-commutative algebras

return their integer type ID here (see section 2.10), or, for `clifford` and `color` objects with different representation labels, a variation thereof (see below). Objects of different type are assumed to commute with each other. Products of such objects are considered as being of composite type, and don't commute with any other non-commutative objects. The commutation properties of a container object don't need to be related to that of its subexpressions. For example, while a `matrix` object A is non-commutative (with a `return_type_tinfo()` of `TINFO_matrix`), an indexed matrix A_{ij} , which is an `indexed` object with a `matrix` as its base expression, is commutative. Symbolic functions (see section 2.6.6) can also be defined as non-commutative. By default, they inherit the commutation properties of their first argument.

GiNaC automatically groups products involving non-commutative factors by their type. For example, the expression

$$\gamma^\mu i T_b e \gamma^\nu T_a$$

containing two `clifford` objects γ^μ and γ^ν , two `color` objects T_a and T_b , and two commutative objects i and e , is rearranged to

$$i e (\gamma^\mu \gamma^\nu) (T_b T_a) \quad (3.1)$$

which is a commutative product (`mul`) of four factors, the last two factors being non-commutative products (`ncmul`) of two `clifford` and two `color` objects, respectively. The entire expression is then of composite type. Note that the relative order of the non-commuting factors has not changed.

This grouping is one of the automatic evaluation rules implemented in `ncmul::eval()`. As a consequence, non-commutative products only contain objects of one single type, or of composite type. In the following, we will refer to single-type products as *strings*. Expression (3.1) thus contains two strings: the Dirac string $\gamma^\mu\gamma^\nu$ and the color string T_bT_a .

Within each string, GiNaC performs class-specific canonicalizations, such as anticommuting γ^5 to the front of Dirac strings. These are implemented by the method `eval_ncmul()` in the respective classes, which is invoked by `ncmul::eval()` as its last step.

After these general remarks we will now discuss the Dirac and color algebras in more detail.

3.4.1 The Dirac Algebra

The matter part of the Standard Model consists of a set of spin- $\frac{1}{2}$ fermions represented by four-component fields $\psi_A(x)$ satisfying the Dirac equation

$$(i\gamma_{AB}^\mu\partial_\mu - m\mathbb{1}_{AB})\psi_B(x) = 0 \quad (3.2)$$

where m is the mass of the field and $\mathbb{1}_{AB} \equiv \delta_{AB}$ is the Kronecker delta which we write as $\mathbb{1}$ here because it is represented by a different class than `tensdelta` in GiNaC.

The objects γ^μ , defined by

$$\gamma_{AB}^\mu\gamma_{BC}^\nu + \gamma_{AB}^\nu\gamma_{BC}^\mu = 2\eta^{\mu\nu}\mathbb{1}_{AC}, \quad (3.3)$$

are called *Dirac matrices* and can be represented by complex 4×4 matrices. γ^μ also behaves as a vector under Lorentz transformations. The set of 16 objects

$$\begin{aligned} &\mathbb{1}, \quad \gamma^0, \quad \gamma^1, \quad \gamma^2, \quad \gamma^3, \\ &\gamma^0\gamma^1, \quad \gamma^0\gamma^2, \quad \gamma^0\gamma^3, \quad \gamma^1\gamma^2, \quad \gamma^1\gamma^3, \quad \gamma^2\gamma^3, \\ &\gamma^0\gamma^1\gamma^2, \quad \gamma^0\gamma^1\gamma^3, \quad \gamma^0\gamma^2\gamma^3, \quad \gamma^1\gamma^2\gamma^3, \\ &\gamma^0\gamma^1\gamma^2\gamma^3 \end{aligned} \quad (3.4)$$

forms a basis of the Clifford algebra $\mathcal{C}(M_4)$ of the four-dimensional Minkowski space M_4 with regard to the metric $\eta^{\mu\nu}$ (in the matrix representation, it also forms a basis of all complex 4×4 matrices). See [Pert 2001] for a discussion of the general definition and the properties of Clifford algebras.

In a free field theory, the equation of motion (3.2) can be derived from the dynamic term

$$\mathcal{L}_F = \bar{\psi}_A(i\gamma_{AB}^\mu\partial_\mu - m\mathbb{1}_{AB})\psi_B$$

in the Lagrangian by standard methods (see, for example [BDJ 2001, chapter 2] and [PeSc 1995, chapter 9]). Its two-point Green function is the fermion propagator

$$iS_F(p, m)_{AB} = \frac{i(\gamma_{AB}^\mu p_\mu + m\mathbb{1}_{AB})}{p^2 - m^2 + i\rho}.$$

Contractions such as $\gamma^\mu p_\mu$ are usually abbreviated using the “slash” notation

$$\not{p} := \gamma^\mu p_\mu$$

so the propagator takes the form



$$\frac{i(\not{p}_{AB} + m\mathbb{1}_{AB})}{p^2 - m^2 + i\rho}. \quad (3.5)$$

Similarly, taking QED as the simplest interacting theory, an interaction term of the form

$$\mathcal{L}_I = -eA_\mu \bar{\psi}_A \gamma^\mu_{AB} \psi_B$$

yields the factor

$$-ie\gamma^\mu_{AB} \quad (3.6)$$

for the Feynman rule of the $\psi\bar{\psi}A$ interaction vertex.

Additionally, the Standard Model of the electroweak interaction contains couplings that are different for the left-handed and right-handed components of Dirac fields. We introduce the projection operators

$$\gamma^L = \frac{1}{2}(1 - \gamma^5), \quad \gamma^R = \frac{1}{2}(1 + \gamma^5),$$

with (in a four-dimensional Minkowski space)

$$\gamma^5 := i\gamma^0\gamma^1\gamma^2\gamma^3 = -\frac{i}{4!}\epsilon_{\mu\nu\rho\sigma}\gamma^\mu\gamma^\nu\gamma^\rho\gamma^\sigma \quad (3.7)$$

having the anticommutation property¹²

$$\gamma^5\gamma^\mu = -\gamma^\mu\gamma^5, \quad \gamma^5\gamma^5 = \mathbb{1}. \quad (3.8)$$

The components of a Dirac field $\psi_A(x)$ are

$$\psi_A^L = \gamma^L_{AB}\psi_B, \quad \psi_A^R = \gamma^R_{AB}\psi_B.$$

The factor for the $\psi\bar{\psi}A$ vertex (where $A = \gamma, Z, W^\pm$) then takes the general form



$$ie\gamma^\mu_{AB}(c_L\gamma^L + c_R\gamma^R) \quad (3.9)$$

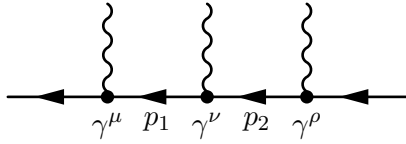
¹² This property is retained even for $D \neq 4$ dimensions (see below).

with coefficients c_L, c_R that depend on the nature of the gauge field A .

Using the Feynman rules (3.5), (3.6), and (3.9) in the given form, with explicit spinor indices, the amplitudes for arbitrary processes could be constructed by simply multiplying all the individual factors for the vertices and propagators. The factors themselves are commutative since the individual components of γ_{AB}^μ are simply complex numbers.

`xloops` applies a slightly different set of Feynman rules that gets by without explicit spinor indices at the cost of some additional, but straightforward, bookkeeping. As the factors in this set of rules contain the non-commutative objects γ^μ , γ^L , γ^R , and slashed momenta in their numerators, one has to keep track of the order they are assembled in.

For a single fermion line, the factors are strung together as they appear along the line. By convention, the line is followed in the direction of particle-number flow, which for electrons, muons, and taus is the direction of negative charge flow (here and in the following we only write out the numerator of the truncated Feynman diagram and omit purely commutative factors like i , e , and the metric tensor $\eta^{\mu\nu}$):



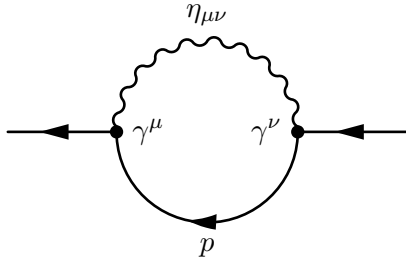
With spinor indices:

$$\gamma_{AB}^\mu (\not{p}_1 + m\mathbb{1})_{BC} \gamma_{CD}^\nu (\not{p}_2 + m\mathbb{1})_{DE} \gamma_{EF}^\rho$$

Without spinor indices:

$$\gamma^\mu (\not{p}_1 + m\mathbb{1}) \gamma^\nu (\not{p}_2 + m\mathbb{1}) \gamma^\rho$$

Virtual gauge boson exchanges involve Lorentz contractions between Dirac matrices that can be simplified using the algebraic property (3.3) of the Dirac matrices (working in D space-time dimensions and the Feynman gauge):



With spinor indices:

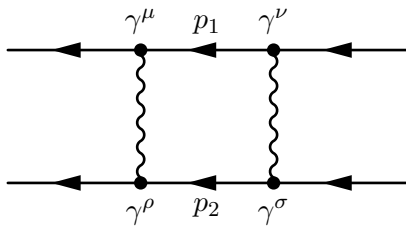
$$\gamma_{AB}^\mu (\not{p} + m\mathbb{1})_{BC} (\gamma_\mu)_{CD}$$

Without spinor indices:

$$\gamma^\mu (\not{p} + m\mathbb{1}) \gamma_\mu = (2 - D)\not{p} + Dm\mathbb{1}$$

Other possibilities are contractions of Dirac matrices with external or loop momenta.

If one has two or more unconnected fermion lines, only those Dirac objects belonging to the same line are multiplied. In the index-free notation it is therefore necessary to distinguish multiple sets of Dirac objects belonging to different fermions. In GiNaC this is achieved by assigning a *representation label* (an integer number) to every Dirac object:



With spinor indices:

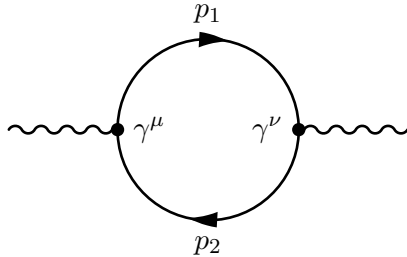
$$\gamma_{AB}^\mu (\not{p}_1 + m\mathbb{1})_{BC} \gamma_{CD}^\nu \gamma_{EF}^\rho (\not{p}_2 + m\mathbb{1})_{FG} \gamma_{GH}^\sigma$$

Without spinor indices:

$$[\gamma_{(1)}^\mu (\not{p}_{1(1)} + m\mathbb{1}_{(1)}) \gamma_{(1)}^\nu] [\gamma_{(2)}^\rho (\not{p}_{2(2)} + m\mathbb{1}_{(2)}) \gamma_{(2)}^\sigma]$$

Objects or strings of objects with different representation labels such as the two brackets in the last expression commute. This expression therefore actually contains two separate Dirac strings.

Finally, a closed fermion loop corresponds to a trace over the Dirac matrices:



With spinor indices:

$$\gamma_{AB}^{\mu}(\not{p}_2 + m\mathbb{1})_{BC}\gamma_{CD}^{\nu}(\not{p}_1 + m\mathbb{1})_{DA}$$

Without spinor indices:

$$\text{Tr}[\gamma^{\mu}(\not{p}_2 + m\mathbb{1})\gamma^{\nu}(\not{p}_1 + m\mathbb{1})]$$

In the case of multiple fermion loops, traces are taken separately for each Dirac string with a different representation label. Traces also appear in the calculation of squared matrix elements.

In four space-time dimensions, or in the absence of axial couplings which involve γ^5 it does not matter at which point in the loop one starts assembling the factors since the trace operation is symmetric with respect to cyclic permutations of the factors in its argument. We will come back to the problems appearing with γ^5 in dimensional regularization later in this section.

The xloops graph generator is responsible for obeying all these rules and for generating the appropriate representation labels and traces.

We can now give a list of the requirements for the implementation of the Dirac algebra in GiNaC:

- objects representing the Dirac matrices $\mathbb{1}$, γ^{μ} , γ^5 , γ^L , and γ^R ,
- objects representing slashed expressions like \not{p} ,
- representation labels for the above objects,
- the ability to construct non-commutative strings of objects with the same label,
- simplifications such as Lorentz contractions, and some degree of canonicalization of strings,
- the Dirac trace operation.

Although we call them Dirac *matrices*, they are not represented by `matrix` objects in GiNaC, but by a separate `clifford` class that is derived from `indexed`, completely independent of any explicit matrix representation (see fig. 3.5). The base expressions and indices of the objects are different for each type of Dirac matrix (γ^{μ} , $\mathbb{1}$, Dirac slash, etc.), as shown in fig. 3.6.

The `clifford` class augments `indexed` by an `unsigned char` attribute that stores the object's representation label (an integer number in the range $[0, 255]$). Because its `return_type()` is `noncommutative` and its `return_type_tinfo()` is the sum of

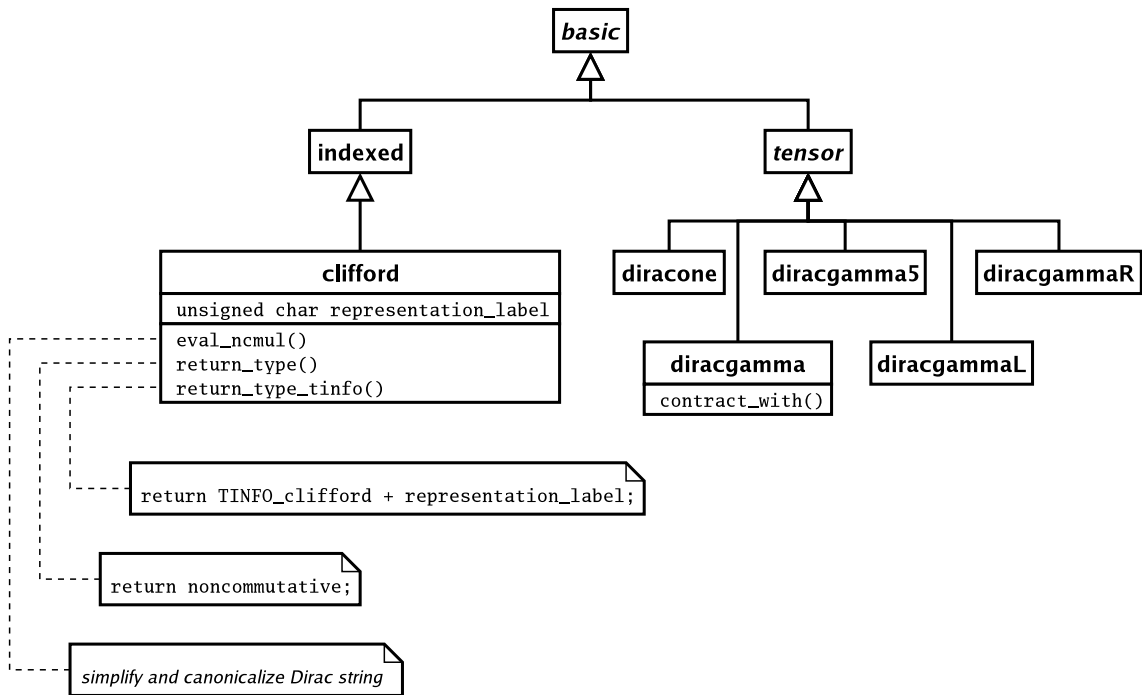


Figure 3.5: Classes implementing the Dirac algebra. The class of Dirac matrices `clifford` augments the `indexed` class by a representation label.

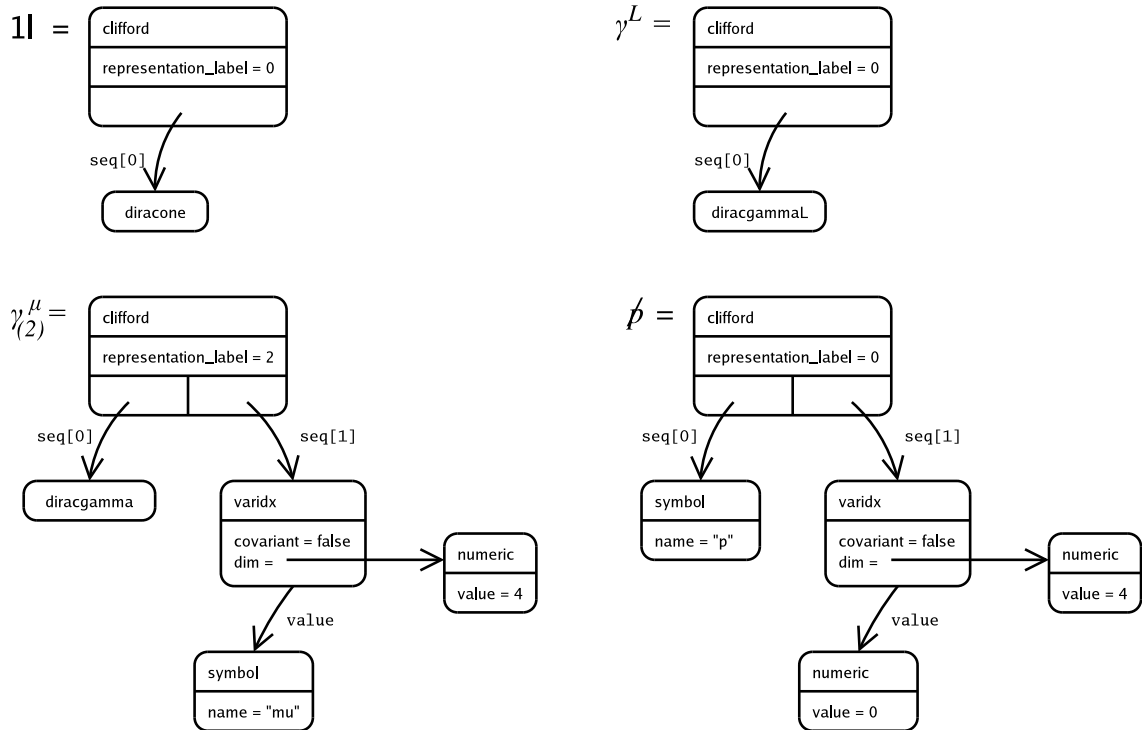


Figure 3.6: Examples for the representation of Dirac matrices in GiNaC

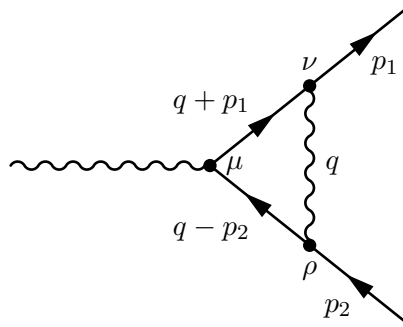
TINFO_clifford and the representation label, GiNaC automatically groups products of Dirac matrices into separate non-commutative Dirac strings for each label.

A couple of subclasses of `tensor` are defined corresponding to the different types of Dirac matrices which are used as the base expressions of `clifford` objects. However, users of the GiNaC library will usually construct Dirac matrices with one of the following helper functions:

- `dirac_ONE(r)`
Returns $\mathbb{1}_{(r)}$, constructing a `clifford` object with the specified representation label, a `diracone` tensor as the base expression, and no indices.
- `dirac_gamma(mu, r)`
Returns $\gamma_{(r)}^\mu$, constructing a `clifford` object with the specified label, a `diracgamma` as the base expression, and the supplied index μ which can be co- or contravariant and of arbitrary numeric or symbolic dimension.
- `dirac_gamma5(r)`, `dirac_gammaL(r)`, `dirac_gammaR(r)`
Return $\gamma_{(r)}^5$, $\gamma_{(r)}^L$, or $\gamma_{(r)}^R$, respectively, constructing a `clifford` object with the specified label, a `diracgamma5`, `diracgammaL`, or `diracgammaR` object as the base expression, and no indices.
- `dirac_slash(p, dim, r)`
Returns $\not{p}_{(r)}$ with `dim` being the dimension of the suppressed contraction index. This is stored as a `clifford` object with the base expression p and a placeholder index of value 0 storing the supplied index dimension.

For all functions, the representation label `r` can be omitted, in which case it is assumed to be 0. As with `lorentz_g()` and related functions, the tensors `diracone` etc. are created as flyweights and shared between Dirac objects.

As an example for using Dirac matrices in GiNaC, consider the one-loop diagram



which, if we assume all vertices to have the same general coupling (3.9), has the numerator structure

$$N = \gamma^\nu (c_L \gamma^L + c_R \gamma^R) (\not{q} + \not{p}_1 + m \mathbb{1}) \gamma^\mu (c_L \gamma^L + c_R \gamma^R) \cdot (\not{q} - \not{p}_2 + m \mathbb{1}) \gamma^\rho (c_L \gamma^L + c_R \gamma^R) \eta_{\rho\nu}.$$

This would be entered in the following way using GiNaC:

```

1 // declare symbols and indices
2 symbol p1("p_1"), p2("p_2"), q("q"), m("m"), D("D");
3 symbol cL("c_L"), cR("c_R");
4 varidx mu(symbol("mu"), D), nu(symbol("nu"), D), rho(symbol("rho"), D);
5
6 // assemble numerator
7 ex N = dirac_gamma(nu) * (cL*dirac_gammaL() + cR*dirac_gammaR())
8     * (dirac_slash(q, D) + dirac_slash(p1, 4) + m * dirac_ONE())
9     * dirac_gamma(mu) * (cL*dirac_gammaL() + cR*dirac_gammaR())
10    * (dirac_slash(q, D) - dirac_slash(p2, 4) + m * dirac_ONE())
11    * dirac_gamma(rho) * (cL*dirac_gammaL() + cR*dirac_gammaR())
12    * lorentz_g(rho.toggle_variance(), nu.toggle_variance());

```

For the sake of simplicity, we will also assume the fermion to be massless ($m = 0$). Lorentz contractions and canonicalizations can then be carried out with

```

13 N = N.subs(m == 0).simplify_indexed();

```

The resulting numerator contains various terms. It can be decomposed into a left-handed part and a right-handed part

$$N = \gamma^L N_L + \gamma^R N_R \quad \Rightarrow \quad N_L = \gamma^L N, \quad N_R = \gamma^R N$$

by multiplying it with γ^L and γ^R , respectively:

```

14 ex NL = (dirac_gammaL()*N).expand();
15 ex NR = (dirac_gammaR()*N).expand();

```

To get a concise result, we will collect N_L and N_R in terms of the various appearing Dirac strings with the aid of a visitor (cf. section 2.18):

```

16 class collect_strings : public visitor,
17                        public add::visitor,
18                        public mul::visitor,
19                        public ncmul::visitor {
20 public:
21     // gather all strings in the set strings
22     void visit(const ncmul & e)
23     {
24         strings.insert(e);
25     }
26
27     // visit all terms of a sum, then collect the sum in terms of the
28     // strings we have gathered
29     void visit(const add & e)
30     {
31         for (const_iterator i = ex(e).begin(); i != ex(e).end(); ++i)
32             i->accept(*this);
33         result = e.collect(lst(list<ex>(strings.begin(), strings.end())));
34     }
35
36     // visit all factors of a product
37     void visit(const mul & e)
38     {
39         for (const_iterator i = ex(e).begin(); i != ex(e).end(); ++i)
40             i->accept(*this);
41     }

```

```

42
43     // return result to caller
44     ex get_result() const { return result; }
45
46 private:
47     set<ex, ex_is_less> strings; // set of strings gathered so far
48     ex result;                 // returned expression
49 };
50
51 // create two visitors and send one each over NL and NR
52 collect_strings NL_visitor, NR_visitor;
53 NL.accept(NL_visitor);
54 NR.accept(NR_visitor);
55
56 // get the collected results and also pull out common scalar terms
57 NL = collect_common_factors(NL_visitor.get_result());
58 NR = collect_common_factors(NR_visitor.get_result());

```

The final result is then

$$\begin{aligned}
N_L &= c_R^3 (2\gamma^L \not{p}_2 \gamma^\mu \not{p}_1 - 2\gamma^L \not{q} \gamma^\mu \not{p}_1 + 2\gamma^L \not{p}_2 \gamma^\mu \not{q} + (2 - D)\gamma^L \not{q} \gamma^\mu \not{q} \\
&\quad - (4 - D)\gamma^L \not{p}_1 \gamma^\mu \not{p}_2 + (4 - D)\gamma^L \not{p}_1 \gamma^\mu \not{q} - (4 - D)\gamma^L \not{q} \gamma^\mu \not{p}_2), \\
N_R &= N_L(L \leftrightarrow R).
\end{aligned}$$

Automatic simplifications of Dirac strings are handled in `clifford::eval_ncmul()`, as explained, while contractions are evaluated in `diracgamma::contract_with()` (since γ^μ is the only Dirac object with a proper index). The exact rules implemented in GiNaC are given in appendix B.2.

To perform the trace operation, the user calls the function `dirac_trace(e, r1, trONE)` which takes the trace over all Dirac strings with the representation label(s) `r1` (either a single label, or a set or list of multiple labels) in the expression `e`. The trace of the identity matrix $\mathbb{1}$ can be specified in `trONE` which defaults to 4.

In $D \neq 4$ dimensions, traces containing γ^5 pose a problem. Obviously, the definition (3.7) is no longer applicable as the Lorentz vector γ^μ doesn't contain exactly four components. It is possible to take the anticommutation property (3.8) as the definition of γ^5 but this turns out to be incompatible with the four-dimensional definition of the trace operation (see [Jege 2001] for a complete discussion).

GiNaC implements the scheme proposed in [Krei 1990] and [KKS 1992] of replacing the trace operation with a linear functional `Tr[]` that only corresponds to the ordinary matrix trace in $D = 4$ dimensions. In general, the `Tr[]` operator projects out the coefficient proportional to $\mathbb{1}$ in the expansion of a Dirac string S in a basis such as (3.4):

$$\text{Tr}[S = a_0\mathbb{1} + (a_1)_\mu\gamma^\mu + \dots] := a_0 \text{Tr}[\mathbb{1}] \quad (3.10)$$

For a string containing γ^5 (because of (3.7) any string can be rewritten with either one or no γ^5), `Tr[]` acts as a projection on the γ^5 term:

$$\text{Tr}[\gamma^5(a_0\mathbb{1} + (a_1)_\mu\gamma^\mu + \dots + a_4\gamma^5 + \dots)] := a_4 \text{Tr}[\mathbb{1}] \quad (3.11)$$

$\text{Tr}[\]$ can be computed in terms of the ordinary four-dimensional trace $\text{tr}[\]$ as follows:

$$\begin{aligned}\text{Tr}[S] &= \text{tr}[S], \\ \text{Tr}[\gamma^5 S] &= \frac{i}{4!} \epsilon_{\mu_1 \mu_2 \mu_3 \mu_4}^{(0123)} \text{tr}[\gamma^{\mu_1} \gamma^{\mu_2} \gamma^{\mu_3} \gamma^{\mu_4} S]\end{aligned}\quad (3.12)$$

where $\epsilon_{\mu_1 \mu_2 \mu_3 \mu_4}^{(0123)}$ is a four-dimensional antisymmetric tensor that corresponds to $\epsilon_{\mu_1[4]\mu_2[4]\mu_3[4]\mu_4[4]}$ in GiNaC.

The trace operation defined by these properties is not cyclic in $D \neq 4$ dimensions for strings containing six or more Dirac matrices and one γ^5 . For example,

$$\eta_{\mu_5 \mu_6} (\text{Tr}[\gamma^5 \gamma^{\mu_1} \gamma^{\mu_2} \gamma^{\mu_3} \gamma^{\mu_4} \gamma^{\mu_5} \gamma^{\mu_6}] - \text{Tr}[\gamma^{\mu_6} \gamma^5 \gamma^{\mu_1} \gamma^{\mu_2} \gamma^{\mu_3} \gamma^{\mu_4} \gamma^{\mu_5}])$$

computed by the program

```

1  symbol D("D");
2  varidx mu1(symbol("mu1"), D), mu2(symbol("mu2"), D), mu3(symbol("mu3"), D),
3      mu4(symbol("mu4"), D), mu5(symbol("mu5"), D), mu6(symbol("mu6"), D);
4
5  ex S = dirac_gamma5() * dirac_gamma(mu1) * dirac_gamma(mu2)
6      * dirac_gamma(mu3) * dirac_gamma(mu4) * dirac_gamma(mu5);
7
8  ex e = lorentz_g(mu5.toggle_variance(), mu6.toggle_variance())
9      * (dirac_trace(S * dirac_gamma(mu6))
10     - dirac_trace(dirac_gamma(mu6) * S));
11
12  cout << collect_common_factors(e.simplify_indexed()) << endl;
    
```

yields

$$8i(D - 4)\epsilon_{\mu_1 \mu_2 \mu_3 \mu_4}$$

which only vanishes for $D = 4$.

The non-cyclicity of the trace implies that one has to choose a specific *reading point* when constructing the Feynman amplitude for a diagram with fermion loops in the presence of axial couplings. The `xloops` graph generator needs to implement rules such as given in [KKS 1992] to ensure correct results for these diagrams.

Unfortunately, the complexity of the Dirac trace algorithm implemented in GiNaC is $\mathcal{O}(N!)$, which limits the length N of strings whose traces can be computed in reasonable time to $N = 12$ (see fig. 3.7). Special provisions for more efficient calculations of products of traces with contracted indices such as those described in [Chis 1963] are not currently in place.

One additional operation available on expressions containing Dirac matrices is `canonicalize_clifford()` which uses the commutation relation (3.3) to bring the elements of Dirac strings into a canonical order. This function will, for example, convert one of $\gamma^\mu \gamma^\nu$ and $2\eta^{\mu\nu} \mathbf{1} - \gamma^\nu \gamma^\mu$ into the other form (depending on the internal ordering of μ and ν), allowing to compare them for equality.

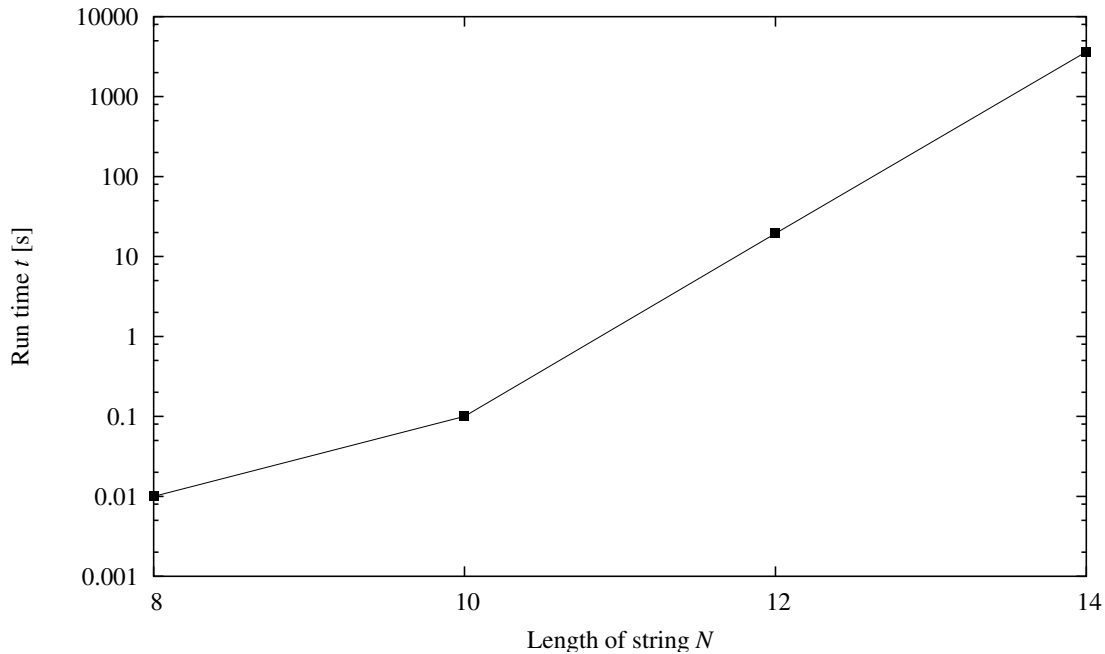


Figure 3.7: Run times for computing $\text{Tr}[\gamma^{\mu_1} \dots \gamma^{\mu_N}]$, depending on the number N of Dirac matrices.

3.4.2 The Color Algebra

The handling of the $SU(3)$ color algebra,¹³ although conceptually quite different from the Dirac algebra, bears remarkable similarities in its implementation in GiNaC.

A multi-component¹⁴ fermion field $\psi_i(x)$ that transforms locally under a symmetry group G represented by a set of infinitesimal generators $T_a, a = \{1, \dots, \dim G\}$ as

$$\psi_i(x) \rightarrow \psi'_i(x) = \left(e^{i\theta_a(x)T_a} \right)_{ij} \psi_j(x)$$

gives rise to a gauge field $A_\mu(x)$ belonging to the Lie algebra of G if the Lagrangian is supposed to be invariant under this transformation. The gauge field can be expanded in terms of the generators:

$$A_\mu(x) = gA_{\mu,a}(x)T_a.$$

The corresponding covariant derivative is then

$$D_\mu = \partial_\mu \mathbb{1}_C - iA_\mu(x),$$

with $\mathbb{1}_C$ being the unit element of the Lie algebra. The field strength tensor is given by

$$F_{\mu\nu,a}(x) = \partial_\mu A_{\nu,a}(x) - \partial_\nu A_{\mu,a}(x) + gf_{abc}A_{\mu,b}(x)A_{\nu,c}(x)$$

¹³ An extension to $SU(N)$ for an arbitrary symbolic N is planned for a future version of GiNaC.

¹⁴ The index i is not a spinor index, which is left out in this discussion.

where f_{abc} are the structure constants of the Lie algebra of G , defining the algebraic properties of the generators T_a by the commutation relation

$$[T_a, T_b] = if_{abc}T_c. \quad (3.13)$$

As in the case of the Dirac algebra, the generators T_a can be represented by matrices $T_{a,ij}$, but this representation is not used by GiNaC, and the matrix indices i, j (whose index set in general differs from the group index a) are not written down explicitly in the Feynman rules.

In QCD, the symmetry group is $G = SU(3)$, $\dim G = 8$ acting on three-component quark spinor fields $\psi_i(x)$, one per quark flavor. From the gauge-covariant fermion part

$$\mathcal{L}_F = \bar{\psi}(i\gamma^\mu D_\mu - m\mathbb{1}_C)\psi$$

and the gauge-field part

$$\mathcal{L}_G = -\frac{1}{4}F_{\mu\nu,a}F_a^{\mu\nu}$$

of the Lagrangian one can derive the expressions for the quark propagator



$$\frac{i(\not{p} + m\mathbb{1})\mathbb{1}_C}{p^2 - m^2 + i\rho} \quad (3.14)$$

and the gluon propagator which in the Feynman gauge reads



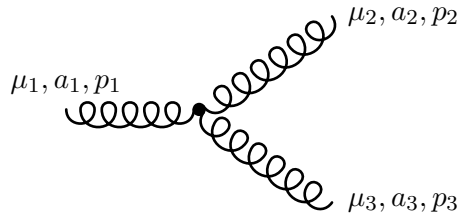
$$\frac{-i\eta_{\mu\nu}\delta_{ab}}{p^2 + i\rho} \quad (3.15)$$

The $\psi\bar{\psi}A$ vertex yields a factor



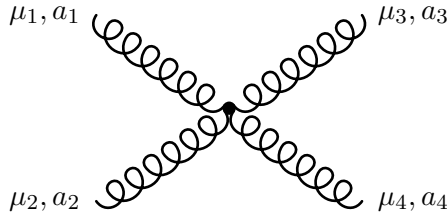
$$ig\gamma^\mu T_a \quad (3.16)$$

Additionally, there are three-gluon



$$gf_{a_1 a_2 a_3} \left(\eta^{\mu_1 \mu_2} (p_1 - p_2)^{\mu_3} + \eta^{\mu_2 \mu_3} (p_2 - p_3)^{\mu_1} + \eta^{\mu_3 \mu_1} (p_3 - p_1)^{\mu_2} \right)$$

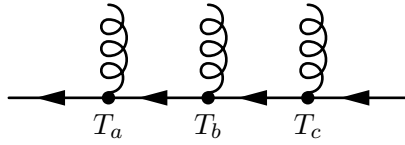
and four-gluon vertices



$$\begin{aligned}
 & -ig^2 (f_{a_1 a_2 b} f_{a_3 a_4 b} (\eta^{\mu_1 \mu_3} \eta^{\mu_2 \mu_4} - \eta^{\mu_1 \mu_4} \eta^{\mu_2 \mu_3}) \\
 & + f_{a_1 a_3 b} f_{a_4 a_2 b} (\eta^{\mu_1 \mu_4} \eta^{\mu_3 \mu_2} - \eta^{\mu_1 \mu_2} \eta^{\mu_3 \mu_4}) \\
 & + f_{a_1 a_4 b} f_{a_2 a_3 b} (\eta^{\mu_1 \mu_2} \eta^{\mu_4 \mu_3} - \eta^{\mu_1 \mu_3} \eta^{\mu_4 \mu_2}))
 \end{aligned}$$

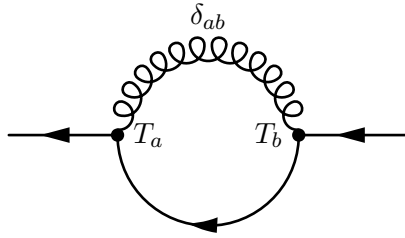
which do not appear in an abelian gauge theory like QED due to the structure constants f_{abc} being zero.¹⁵

As with the Dirac matrices, the factors of a Feynman amplitude have to be multiplied following quark lines (we will omit the Dirac structure in the following examples):



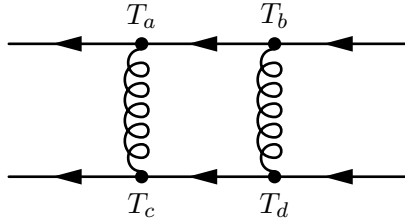
$$T_a \mathbb{1}_C T_b \mathbb{1}_C T_c = T_a T_b T_c$$

There also appear contractions between objects carrying color indices ($T_a, f_{abc}, \delta_{ab}$), such as



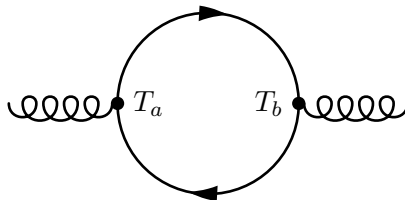
$$T_a \mathbb{1}_C T_b \delta_{ab} = T_a T_a = \frac{4}{3} \mathbb{1}_C$$

Multiple quark lines are distinguished by color matrices carrying different representation labels:



$$[T_a^{(1)} \mathbb{1}_C^{(1)} T_b^{(1)}] [T_c^{(2)} \mathbb{1}_C^{(2)} T_d^{(2)}]$$

and closed quark loops amount to a trace over the generators T_a :



$$\text{Tr}[T_a T_b]$$

¹⁵ They do appear in the non-abelian electroweak theory but do not carry a special Dirac structure which is why we omitted them in our discussion in section 3.4.1.

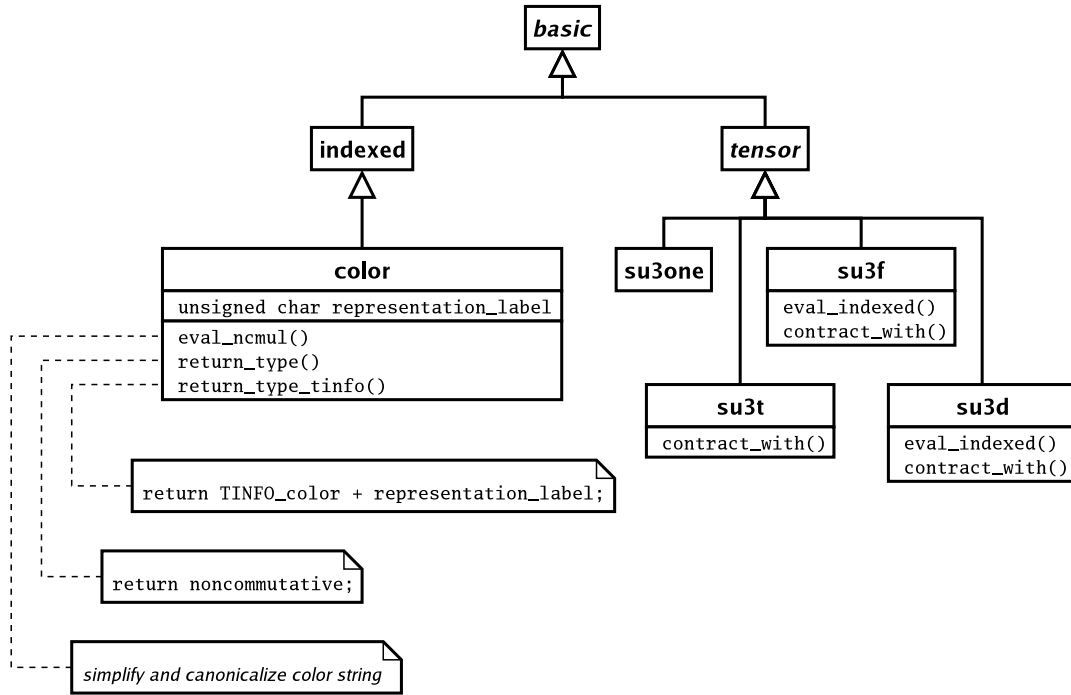


Figure 3.8: Classes implementing the $SU(3)$ color algebra

The features required for the color algebra in GiNaC are thus:

- objects representing the non-commutative generators T_a and the unit $\mathbb{1}_C$, and the commutative δ_{ab} and the antisymmetric structure constants f_{abc} ; for completeness, GiNaC also provides the symmetric structure constants d_{abc} defined by

$$T_a T_b + T_b T_a = \frac{1}{3} \delta_{ab} + d_{abc} T_c$$

- representation labels for the T_a and $\mathbb{1}_C$,
- the ability to construct non-commutative strings of T_a and $\mathbb{1}_C$,
- simplifications such as contractions between T_a , f_{abc} , d_{abc} , and δ_{ab} ,
- color traces.

Fig. 3.8 shows the classes implementing the color algebra, which look very similar to those for the Dirac algebra (cf. fig. 3.5). The `color` class which stores the representation label is only used for representing the non-commutative T_a and $\mathbb{1}_C$.¹⁶ The structure constants f_{abc} and d_{abc} are stored in normal `indexed` objects, and δ_{ab} is just the ordinary Kronecker delta tensor already described in section 3.2.2. All color indices have dimension 8.

The color algebra objects are constructed by the following helper functions:

¹⁶ $\mathbb{1}_C$ commutes with all T_a but this is handled separately in `color::eval_ncmul()`. To GiNaC, $\mathbb{1}_C$ is a non-commutative object.

- `color_ONE(r)`
Returns $\mathbb{1}_C^{(r)}$, constructing a `color` object with the specified representation label, an `su3one` tensor as the base expression, and no indices.
- `color_T(a, r)`
Returns $T_a^{(r)}$, constructing a `color` object with the specified label, an `su3t` as the base expression, and the supplied index.
- `color_f(a, b, c)`
Returns f_{abc} as an `indexed` object with an `su3f` as the base expression, the symmetry $-(0, 1, 2)$, and the three specified indices.
- `color_d(a, b, c)`
Returns d_{abc} as an `indexed` object with an `su3d` as the base expression, the symmetry $+(0, 1, 2)$, and the three specified indices.
- `color_h(a, b, c)`
Returns the sum $h_{abc} := d_{abc} + if_{abc}$.

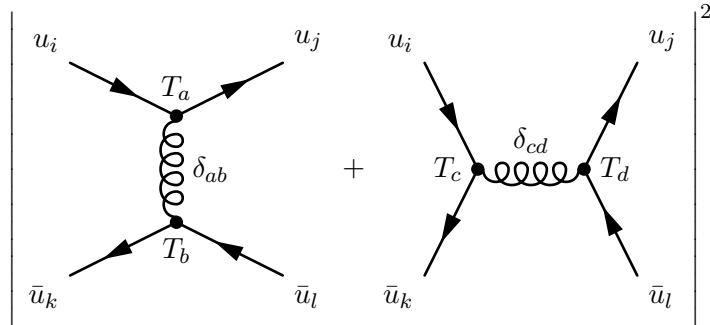
The simplification rules implemented for color algebra objects in GiNaC are listed in appendix B.3. In contrast to the Dirac algebra, the rules for the color algebra deal mostly with contractions between the different types of objects.

Traces of color strings are computed by the function `color_trace(e, r1)` according to the following rules:

- $\text{Tr}[\mathbb{1}_C] = 3,$
- $\text{Tr}[T_a] = 0,$
- $\text{Tr}[T_a T_b] = \frac{1}{2} \delta_{ab},$
- $\text{Tr}[T_a T_b T_c] = \frac{1}{4} h_{abc},$
- $\text{Tr}[T_{a_1} \dots T_{a_n}] = \frac{1}{6} \delta_{a_{n-1} a_n} \text{Tr}[T_{a_1} \dots T_{a_{n-2}}] + \frac{1}{2} h_{a_{n-1} a_n k} \text{Tr}[T_{a_1} \dots T_{a_{n-2}} T_k].$

The last rule allows computing traces of four or more elements by recursively reducing them to traces of shorter strings. As fig. 3.9 shows, this is roughly an $\mathcal{O}(2^N)$ operation.

Concluding this chapter with an example, consider the process $u\bar{u} \rightarrow u\bar{u}$ on the tree level. The squared matrix element contains an interference term between the s - and t -channel contributions



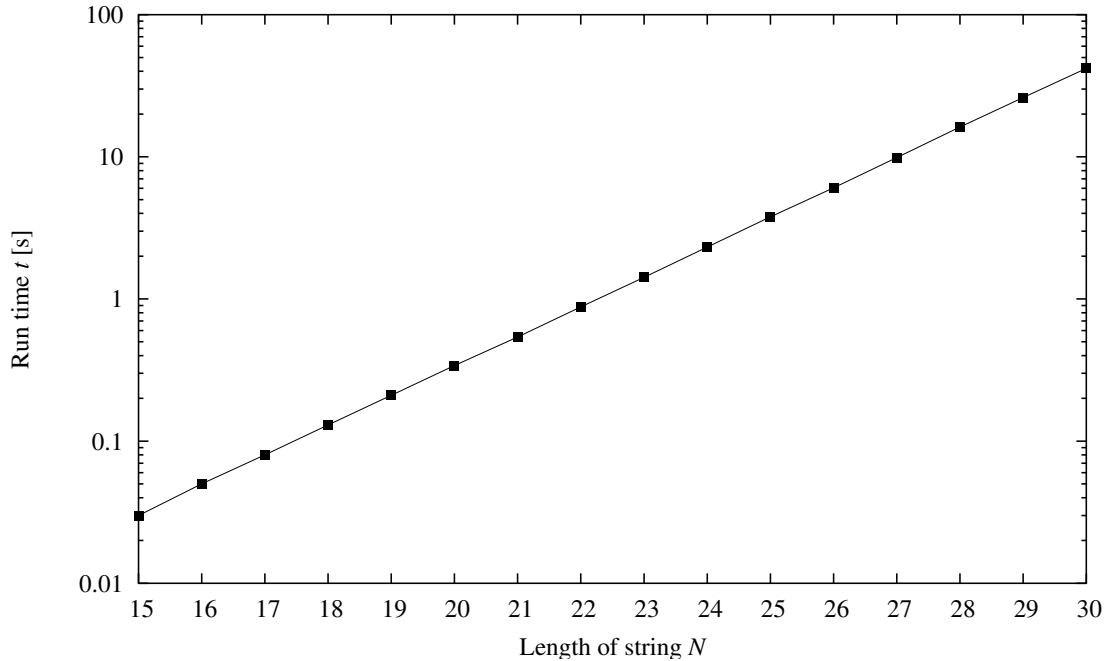


Figure 3.9: Run times for computing $\text{Tr}[T_{a_1} \dots T_{a_N}]$, depending on the number N of color matrices.

which receives a color factor by averaging over the initial colors and summing over the final colors equal to

$$\begin{aligned}
 C &= \left(\frac{1}{3}\right)^2 \sum_{ijkl} T_{a,ji} T_{a,kl} T_{c,ki}^* T_{c,jl}^* \\
 &= \frac{1}{9} \text{Tr}[T_a T_c T_a T_c]
 \end{aligned}$$

which can be computed using GiNaC with

```

1 idx a(symbol("a"), 8), c(symbol("c"), 8);
2 ex C = color_trace(color_T(a)*color_T(c)*color_T(a)*color_T(c)) / 9;
3 cout << C.simplify_indexed() << endl;

```

yielding $C = -\frac{2}{27}$ ($= \frac{1}{N^2}(C_F - \frac{1}{2}C_A)C_F \text{Tr}[\mathbb{1}_C] = \frac{1-N^2}{4N^3}$ for $N = 3$).

4 GiNaC – Status and Outlook

Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get “right” the first time.

Erich Gamma et al., Design Patterns [GHJV 1995]

When you do things right, people won't be sure you've done anything at all.

Futurama Episode 3.20 “Godfellas”

In the past five years, GiNaC has reached a mature state and offers all the features required for the reimplementations of `xloops` carried out in [Baue 2000], [BaDo 2002], and [Do 2003]. It is also efficient and flexible enough to be used by other researchers for projects in physics and other fields. For example, in [BKK 2002] a program has been developed based on GiNaC for the automatic calculation of counter terms for multi-loop diagrams in massless Quantum Field Theory. A two-loop calculation of MSSM corrections to the relation between the pole mass and the running mass of the b quark performed with GiNaC has been presented in [BeSh 2004].

The table below lists some other projects using GiNaC that we are aware of.

Name	Maintainer	Description
nestedsums	S. Weinzierl	Library for symbolic expansion of certain classes of transcendental functions, used by <code>xloops</code> [MUW 2002] [Wein 2004a]
gTybalt	S. Weinzierl	Combines GiNaC with ROOT and the TEX-MACS editor into an interactive computer algebra system [Wein 2004b]
feelfem	H. Fujio, NEC Japan	Finite Element Method code generator
PURRS	R. Bagnara, Univ. of Parma	Library for solving recurrence relations [BaZa 2004] (http://www.cs.unipr.it/purrs/)
MBDyn	P. Mantegazza, Politecnico di Milano	Multibody dynamics analysis software (http://www.mbdyn.org)
Ecco	Kai Ludwig, Univ. of Tübingen	Compiler for the simulation of electrochemical reaction kinetics, part of the EChem++ package (http://echempp.sf.net)

For a comparison of GiNaC to other symbolic manipulators in terms of efficiency and memory usage, we refer the reader to [Krec 2002, section 5.1].

There is always room for improvement, of course. In the following sections we will take up some areas in which further development may be worthwhile.

4.1 Efficient Allocation of Small Objects

As a consequence of the fine granularity of objects in GiNaC, a large amount of small objects are continually allocated and deallocated when working with expressions. To give an impression of the “smallness” of GiNaC’s objects: GCC/i386 allocates 20 bytes for each `basic` object, 24 bytes minimum for each `numeric` object (large numbers take up more space), $36 + 8n$ bytes minimum for each `add` with n terms (not counting the space taken up by the terms), and 48 bytes minimum for a `symbol` (plus the space required to store the symbol’s name).

Of course, each allocation and deallocation request incurs some overhead, both in CPU time and in memory requirement, that for the most part depends on the implementation of memory management in the C++ run-time library and the underlying operating system. For each `new` operation, the C++ memory allocator has to find a suitable free memory block (possibly requesting more storage space from the operating system) and mark it as used. A subsequent call to `delete` needs to relinquish the block to the memory allocator. Assuming an 8-byte overhead for each block (enough for a 4-byte block size and a link pointer) that is typical for simple memory allocation implementations, a GiNaC program would end up wasting about 20% of its available memory, not to mention the extra CPU cycles used up in the process.

Although GiNaC employs lots of techniques to decrease overall memory usage by sharing objects between expressions, as detailed in section 2.9, and modern C++ compilers now do a better job on small-object allocation than they did years ago when they inherited the `malloc()/free()`-based memory management scheme from C that is typically optimized for larger memory blocks (at least several KB), there is potentially some room for improving the general performance of GiNaC here.

A custom memory allocator could be implemented in GiNaC by overloading the `new` and `delete` operators in the `basic` class. All allocations and deallocations of algebraic objects would then use the new routines. The hard part is of course writing and benchmarking memory management algorithms that are optimized to the usage patterns in GiNaC.

The Loki library [Alex 2001] contains a complete and portable implementation of a small-object memory allocator that is supposedly faster than that offered by most C++ compilers. At least for GCC 3.x, this claim is demonstrably false. Utilizing the Loki allocator with GiNaC led to a dramatic decrease in overall performance (up to 50% increase in run time for the standard GiNaC benchmark suite).

4.2 Virtual Memory

The implementation of expressions in GiNaC as a tree of pointers to subexpressions assumes that each object is directly addressable. While this guarantees fast access to all objects, it also means that the maximum expression size is limited by the address space

of the CPU running the code.¹

It is not uncommon nowadays for computers to be equipped with several gigabytes of RAM, but a 32-bit architecture is typically still limited to 2^{32} addressable memory locations. While this is more than enough for the algorithms used in `xloops`, for some applications in particle physics whose calculation procedures produce intermediate results with billions of terms it is far from sufficient.

One of the key features of FORM is its ability to work with expressions that exceed the addressable memory space by buffering them in disk files. This works very well since FORM internally uses flat data structures for representing expressions (at the cost of restricting the possible set of expressions it can handle), and mostly algorithms traversing them in a linear fashion.

A similar scheme could be implemented in GiNaC by making certain algebraic classes like `add` or `mul` (or even `ex`) act as proxies to data that is stored on disk and retrieved when subexpressions are accessed via `op()`. Whether this is feasible remains debatable. The intrinsically hierarchical structure of expressions would require clever storage and access mechanisms (basically, a hierarchical database) to achieve satisfactory performance. A GiNaC expression is not a simple stream of bytes that can be directly written to disk.

Although converting expressions to and from byte streams is possible with GiNaC's object persistence system which we described in section 2.15, it is not particularly suited for this task, at least not in its current implementation. The construction of the `archive_node` tree actually takes up even more temporary memory space, and retrieving object attributes by name is not exactly the fastest possible way of data access. GiNaC's archiving system was designed with data exchange and long-term storage in mind, not for swapping expressions to disk on the fly.

Since 64-bit architectures are expected to enter the commodity PC market within the next two years, we don't see GiNaC's limitation as the most pressing issue. With a 64-bit address space, one could again delegate the problem of virtual memory management to the underlying operating system, as GiNaC has done in the past.

4.3 Expression Input

While GiNaC offers a comprehensive dynamic mechanism for printing expressions, with the ability to add user-defined algebraic classes and output formats (see section 2.14), the facilities for expression input are somewhat impoverished in comparison.

The YACC-based input parser described in section 2.22 is completely static. There is neither a way to define new input formats nor to make user-added algebraic classes parseable.² Also, since due to its C roots, YACC only performs textual manipulations on source code fragments that implement the parser actions, its use in the context of a C++ library feels somewhat "alien".

GiNaC's expression input parser should eventually be replaced by an implementation that

¹ It may be further limited by parts of the address space being reserved by the operating system.

² A workaround is to parse instances of user-defined classes as symbols or symbolic functions, and substituting them by the proper objects in the parsed expression.

stores the grammar in dynamic data structures, thus enabling new classes to add their own parsing rules. A possible candidate for such an implementation is SPIRIT [Guzm 2002], a C++ template library that allows writing rules in an EBNF-like form using C++ syntax, hooking callback functions to the rules, and parsing an input data stream according to the supplied rule set. It also has some support for creating and modifying rules at run time.

4.4 Graph Algorithms

As explained in section 2.7, the run-time structure of a GiNaC expression is that of a directed acyclic graph, but to the outside, for example in the `op()` method or the tree traversal iterators from section 2.17, GiNaC presents an expression as a tree, corresponding to its logical structure.

While the idea is to hide implementation details like subexpression sharing from the user, this is suboptimal from a performance point of view. On the one hand, operations on objects should be done only once, even if the object is referenced multiple times in the same expression. For example, expanding an expression such as

$$e = af(x) + bf(x) + cf(x)$$

containing multiple instances of a function $f(x)$ into a power series in x will call the (possibly costly) series expansion routine of f three times, regardless of whether there might be only one actual `function` object in the expression.

On the other hand, algorithms that employ tree traversal to transform expressions have a tendency to “rip apart” the expression graph structure unless they are implemented very carefully. Consider, for instance, the following seemingly obvious implementation of `power::subs()` (leaving out some details which are unnecessary to this discussion):

```
1  ex power::subs(const exmap & m) const
2  {
3      // recursion into subexpressions
4      ex t = power(basis.subs(m), exponent.subs(m));
5
6      // check for substitution of this node
7      return t.subs_one_level(m);
8  }
```

This works, but creates a new `power` object for every invocation of `subs()`, even if the operation didn’t do any changes to the expression. The actual implementation of `power::subs()` in GiNaC therefore looks something like this, avoiding the creation of a new object unless the basis or exponent have changed:

```
1  ex power::subs(const exmap & m) const
2  {
3      ex new_basis = basis.subs(m);
4      ex new_exponent = exponent.subs(m);
5
6      if (new_basis == basis && new_exponent == exponent)
7          return this->subs_one_level(m);
8      else
9          return power(new_basis, new_exponent).subs_one_level(m);
10 }
```


Algorithms like these might benefit from changing from tree traversals to proper graph traversals that visit each node at most once. This would of course involve keeping track of the nodes that have already been visited, and remembering the results of the operations performed on them (since GiNaC's algorithms are non-mutating). The `map()` method is one possible place where a graph traversal could be implemented. However, whether graph traversals actually bring about any benefits in the light of the run-time and memory overhead caused by the node-tracking would have to be examined carefully.

4.5 Expression Construction

GiNaC overloads the arithmetic operators `+`, `-`, `*`, `/`, and the `+` and `-` prefix operators to allow writing down symbolic expressions in the standard C++ notation.

The C++ arithmetic operators are, however, strictly binary, i.e. they only work on two operands at a time. For GiNaC, this means that expressions are constructed part-by-part, with a lot of intermediate expressions being created and deleted along the way. For example, the single line

```
1  ex e = a - b - c - d;
```

sets off the following series of operations:

1. `operator-(a,b)`
first constructs a temporary `mul` object holding $(-1) \cdot b$, then an `add` object from the terms a and $-b$, which is canonicalized and returned, and deletes the temporary `mul` object,
2. `operator-(a-b,c)`
constructs a `mul` object holding $(-1) \cdot c$, then an `add` from the terms $a - b$ and $-c$, which is canonicalized and returned, and deletes the `mul` and the intermediate $a - b$ `add` object,
3. `operator-(a-b-c,d)`
constructs a `mul` object holding $(-1) \cdot d$, then an `add` from the terms $a - b - c$ and $-d$, which is canonicalized and returned, and deletes the `mul` and the intermediate $a - b - c$ `add` object.

There is clearly a lot of redundancy here. One possible solution is to use the expression templates technique mentioned in section 2.2 to generate a compile-time representation of the expression to be constructed using nested template classes, which compiles into optimized code for creating the expression at run time. High-performance scientific C++ matrix libraries such as BLITZ++ [Veld 2002] use this technique to allow writing down matrix arithmetic in the usual C++ notation without generating intermediate objects or multiple loops.

As an experiment, we have implemented a different, but related technique in an attempt to cut down on the number of temporary objects in the creation of GiNaC expressions, that doesn't rely that much on templates.

Expression	Change	Expression	Change
$x + y$	+56%	$x \cdot y$	+43%
$x - y$	-26%	x/y	+3%
$x + y + 4$	+25%	$x \cdot y \cdot 4$	+24%
$x + y - 4$	-27%	$x \cdot y/4$	-13%
$x - y + 4$	-31%	$x/y \cdot 4$	-1%
$x - y - 4$	-48%	$x/y/4$	-25%
$a + b + c + d + e + f$	-11%	$a \cdot b \cdot c \cdot d \cdot e \cdot f$	-11%
$a - b - c - d - e - f$	-58%	$a/b/c/d/e/f$	+15%

Table 4.1: Changes in run time for the creation of expressions using temporary objects to collect terms of sums and factors of products, compared to the default arithmetic operators of GiNaC 1.2.

The basic idea is to make `operator+(ex,ex)` and `operator-(ex,ex)` return not an expression but an instance of a special class `te_add` that stores the terms of the sum as a vector of expression pairs ready for the creation of an `add` object. `te_add` in turn overloads the `+` and `-` operators to collect any additional terms. It is not until the `te_add` is used in a context where an expression (an `ex` object) is expected that it finally creates an `add` from its assembled terms. This is achieved by providing a casting operator `operator ex()` for the `te_add` class.

With this intermediate class, our example runs as follows:

1. `operator-(a,b)`
constructs and returns a `te_add` object holding the two pairs $(a, 1)$ and $(b, -1)$,
2. `te_add::operator-(c)`
appends the pair $(c, -1)$ to the existing `te_add`,
3. `te_add::operator-(d)`
appends the pair $(d, -1)$ to the existing `te_add`,
4. `te_add::operator ex()`
invoked from the `ex(const ex &)` copy constructor, creates an `add` object from the collected pairs $\{(a, 1), (b, -1), (c, -1), (d, -1)\}$, which is canonicalized and returned.

A similar class, `te_mul` handles products and the `*` and `/` operators. There are also some specializations of the operators in place to provide additional shortcuts when symbols or numbers are used as operands.

Tests have shown mixed results with this approach. In general, it works better with subtraction and division, and with expressions containing sums and products of more than three terms. Table 4.1 lists the changes in run time for the creation of certain sample expressions.

Based on these observations, we modified the implementation to only employ `te_add` and `te_mul` objects for the `-` and `/` operators, and compared the run times of the tests in the standard GiNaC benchmark suite. All tests run faster with the modified arithmetic operators, typically by about 10%, but by up to 20% in some cases (in particular the matrix-based tests).

While the results look very promising, there is one fundamental problem with this implementation: Methods defined for expressions can't directly be invoked on the introduced temporary objects. One is no longer able to, for example, write

```
1 ex frac = (a/b).normal();
```

because the result of `a/b` is a `te_mul`, not an `ex`. The `te_add` and `te_mul` classes would have to reimplement every single public member function of `ex`, forwarding them to the `ex` class after creating a proper expression from the collected list of terms. Also, having specialized operators for, e.g. `ex-ex`, `numeric-numeric` (these two are in GiNaC 1.2), `ex-numeric`, and `ex-int` introduces ambiguities for other combinations of classes such as `numeric-int` (which could resolve to either `ex-int` or `numeric-numeric` via the implicit conversions `numeric→ex` and `int→numeric`).

For these reasons, the described optimized expression construction mechanism has not yet been officially included in GiNaC.

4.6 Multithreading

With the increasing prevalence of affordable multiprocessing-capable computers in the form of multi-CPU machines and machines with one or many multi-core CPUs, it makes sense to consider the use of GiNaC in multithreaded programs.

Programming with multiple threads of execution requires a high amount of diligence in coordinating the access of threads to shared data. Due to the object-sharing techniques employed by GiNaC (described in section 2.9), such coordination is required even when threads operate on disjunctive sets of expressions.

Currently, GiNaC has no support for multithreading, but such support should be relatively easy to add, thanks to the abdication of global state affecting calculations, and the general non-mutability of expressions in GiNaC. The three key areas of the library that would have to be revised are:

1. The reference counting mechanism needs to implement proper locking to ensure the atomicity of its operations. The state of the reference counters needs to be consistent even when multiple threads add or remove references to the same object simultaneously.
2. Similar locking needs to be provided for the places where GiNaC uses static variables for caching data. For example, Bernoulli numbers (which are used in the calculation of dilogarithms and the Zeta function) are computed using a recursion relation, so to speed up the process all previously calculated numbers of the sequence are stored in a table. Access to this table would need to be protected so that either one thread is modifying it, or one or more threads are reading from it at the same time (single-writer/multiple-reader locking).
3. The fusing of subexpressions during comparisons makes the use of C++ pointers or references to algebraic objects that are part of an expression even more dangerous

than in the single-threaded case, because these references could become invalid as soon as another thread operates on an expression that happens to contain the same object. But direct references to objects are required every time a class-specific member function is to be invoked. For this purpose, GiNaC provides a template function `ex_to<T>(ex)` returning a reference to the object of class `T` pointed to by the specified expression.³ For instance, the `numeric` class has a method `to_int()` that converts the stored number to the C++ `int` type, if possible:

```
1 ex e = ...
2 if (is_exactly_a<numeric>(e)) {
3     int i = ex_to<numeric>(e).to_int();
4     // do something with i
5 }
```

In a multithreaded application it might occur that the reference returned by `ex_to<numeric>()` gets invalid before the invocation of `to_int()` because of the `numeric` object getting deleted by another thread, thus leading to undefined behavior. The only solution to this problem is to require that *all* references to algebraic objects go through the reference-counting mechanism. `ex_to<T>()` would then have to return a `ptr<T>` (see section 2.9.1) or a similar object.

Of course, all of this causes additional overhead, especially the locking required for the reference counting. Whether it is still worthwhile to be able to use GiNaC in multithreaded applications remains to be studied.

4.7 Polynomial Factorization

One important algebraic algorithm that is missing in GiNaC is the ability to decompose arbitrary multivariate polynomials into a product of irreducible factors (i. e. polynomials that can't themselves be represented as a product of two or more polynomials).

Such a factorization not only often leads to a more compact and readable representation of an expression. It is also a prerequisite for other important algorithms such as the Risch algorithm for indefinite integration [GCL 1992, chapter 12] and partial fraction decomposition of rational functions.

The standard algorithm for factoring polynomials with rational coefficients, used by many modern computer algebra systems, is the one presented by Mark van Hoeij in [Hoei 2002]. It replaces the Berlekamp-Zassenhaus algorithm [Zass 1969] which for certain sets of input polynomials has a time complexity that is exponential in the degree of the polynomials.

An implementation of polynomial factorization for GiNaC need not be done entirely from scratch. The `GTyBALT` system [Wein 2004b] provides a routine for factoring univariate polynomials in GiNaC by passing them to the NTL library [Shou 2002] which implements the van Hoeij method, converting the input and output polynomials to and from the NTL representation in the process. Based on this routine, multivariate polynomials could then be factored by projecting them to the univariate domain and reconstructing the multivariate solution from the univariate one by Hensel lifting [GCL 1992, chapter 6].

³ Due to lack of run-time type checking in this function, it is effectively an empty operation.

4.8 Algebraic Capabilities

Apart from polynomial factorization there are many other algebraic algorithms missing from GiNaC 1.2 which are generally considered as belonging to the standard repertoire of a computer algebra system. As they are not required by `xloops` we have not yet made any efforts at implementing them.

These algorithms include:

- Limits

For analytic functions, limits at finite points like $\lim_{x \rightarrow 0} \frac{\sin x}{x}$ can be computed by series expansion (which yields $1 + \mathcal{O}(x^2)$ in this case). But for other classes of functions, or limits at infinity, different techniques are needed (see [Grun 1996] for an overview).

- Symbolic summation and series

Except for dummy indices, GiNaC has no notion of symbolic sums such as $\sum_{i=0}^{n-1} i^2 = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}$. For definite sums where the (bounded) summation interval is explicitly given, these could of course be summed explicitly although this may not be the most efficient way of computing them. There exist algorithms for obtaining closed expressions for various types of definite and indefinite sums [Gosp 1978] [PWZ 1997].

- Integration

It may appear absurd, but although GiNaC was created for developing a program whose main purpose is the calculation of certain integral functions, it completely lacks any symbolic or numeric integration capabilities. For numeric integrations, there are of course many established packages to choose from, such as VEGAS [Lepa 1978] and PARINT [DGBEG 1996], which are also used by `xloops`. The complexity of symbolic integration algorithms ranges from trivial (polynomials) to highly involved, depending on the nature of the integrand [GCL 1992, chapters 11 and 12]. GiNaC 1.3 will feature some basic symbolic and numeric integration methods.

- Gröbner bases

Gröbner bases (bases of ideals in polynomial rings) [GCL 1992, chapter 10] have important uses in many symbolic algorithms including simplification of expressions and solving systems of equations. Their integration in GiNaC has not yet been investigated.

- Solving non-linear and differential equations

Presently, GiNaC can only solve linear systems of equations via its `matrix` class. Solving differential equations symbolically is still a very active field of research but for applications in physics, solutions obtained by numerical methods are often sufficient.

5 xloops One-Loop Integral Functions

Den ernsten Leser bitten wir, die letzten Witzeleien dieses Handbuchs zu entschuldigen und versichern ihm, daß es ab hier nichts mehr zu lachen gibt.

From the CBM 8050 manual

The central part of `xloops` is a library of functions for calculating one-loop one-, two-, and three-point, and two-loop two-point Feynman integrals.¹ In this chapter, we will mainly focus on the one-loop functions as they are the most complete. A full description of the two-loop algorithms can be found in [Do 2003].

5.1 Loop Integrals

The expression obtained for the truncated Green function of a single Feynman diagram with one or more closed loops in the Standard Model and related theories can be written in the general form

$$G_{\text{trunc}}(q_1, q_2, \dots) = \int \frac{d^D l_1}{(2\pi)^D} \int \frac{d^D l_2}{(2\pi)^D} \dots \frac{N(l_1, l_2, \dots, q_1, q_2, \dots)}{\prod_i^n P_i(Q_i^2, m_i^2)} \quad (5.1)$$

with each

$$P_i(Q_i^2, m_i^2) = Q_i^2 - m_i^2 + i\rho \quad (5.2)$$

corresponding to the propagator of a particle with mass m_i on an internal line carrying the momentum Q_i which is a linear combination of the loop momenta l_m and the external momenta q_n . Each UV-divergent loop integral is taken over $D = 4 - 2\varepsilon$ space-time dimensions (for finite integrals, one can set $D = 4$ before or after the integration).

The numerator N of the integrand consists of components and squares of internal and external momenta, scalar products of momenta, numeric constants including coupling strengths and mixing angles, and elements of special algebras like Dirac matrices. By moving all objects that do not depend on the integration variables out of the integrals, the expression can be decomposed into a linear combination of *tensor integrals*

$$T_{(n)}^{\mu_1 \mu_2 \dots; \nu_1 \nu_2 \dots} = \int d^D l_1 \int d^D l_2 \dots \frac{l_1^{\mu_1} l_1^{\mu_2} \dots l_2^{\nu_1} l_2^{\nu_2} \dots}{\prod_i^n P_i(Q_i^2, m_i^2)}. \quad (5.3)$$

The *tensor degree* of such an integral is defined as the total number of Lorentz indices appearing in the numerator of the integrand.

¹ One-loop four-point and two-loop three-point functions are still under development.

The one-loop n -point tensor integrals

$$T_{(n)}^{\mu_1\mu_2\dots} = \int d^D l \frac{l^{\mu_1} l^{\mu_2} \dots}{\prod_i^n P_i(Q_i^2, m_i^2)}. \quad (5.4)$$

have been solved in a general way by 't Hooft, Passarino, and Veltman in the 1970s [PaVe 1979, tHVe 1979]. The conventional technique for evaluating these integrals can be outlined as follows:

1. Due to the Lorentz covariance and symmetry of the integral with respect to the tensor indices μ_k , eq. (5.4) can be decomposed into a linear combination of symmetric tensors constructed from the Minkowski metric tensor $\eta^{\mu\nu}$ and the external momenta q_n^μ . For example, the two-point integral of degree 2 with one external momentum q^μ is decomposed into

$$T_{(2)}^{\mu_1\mu_2} = \eta^{\mu_1\mu_2} T_{00} + q^{\mu_1} q^{\mu_2} T_{11}. \quad (5.5)$$

2. The coefficient functions of the tensor decomposition (T_{rs} in the above example) are further reduced to scalar integrals by expressing squares of l and scalar products of l with the external momenta in terms of the propagators P_i , cancelling propagators in the integrands as far as possible, and solving the resulting system of linear equations for the coefficients.
3. The integrands of the remaining scalar integrals

$$S_{(n)} = \int d^D l \frac{1}{\prod_i^n P_i}. \quad (5.6)$$

are transformed into rational functions quadratic in l by *Feynman parametrization*

$$\frac{1}{\prod_i^n P_i} = (n-1)! \int_0^1 dx_1 \dots dx_n \frac{\delta(1 - \sum_i^n x_i)}{(\sum_i^n P_i x_i)^n} \quad (5.7)$$

and shifting the loop momentum in the denominator.

4. By performing the *Wick rotation*

$$l^0 \rightarrow il^0 \quad (5.8)$$

and changing the variable of the l^0 integration

$$\int_{-\infty}^{\infty} dl^0 \rightarrow i \int_{-\infty}^{\infty} dl^0 \quad (5.9)$$

the integrand is made spherically symmetric in Euclidean space and the integration can be split into a one-dimensional integral over the radius l , and the constant integral over the surface of the D -dimensional unit sphere:

$$\int d^D l f(l^2) = \frac{2\pi^{D/2}}{\Gamma(D/2)} \int_0^\infty dl l^{D-1} f(l^2). \quad (5.10)$$

5. The remaining integrations over l and the Feynman parameters x_i can then be performed analytically.

Trying to apply this method to the general case of two-loop tensor integrals one runs into problems, especially for massive integrals with more than two external momenta:

- The tensor reduction in step 2 can't be done completely since there are not enough propagators available for rewriting all scalar products, leaving the linear system of equations to be solved under-determined.
- Most integrations can no longer be performed analytically. For divergent integrals, one has to find suitable subtraction terms to obtain numerically integrable, finite integrals.
- The Feynman parametrization in step 3 introduces additional integration variables, which reduces the performance and stability of the numeric integration.

In the light of these problems, several alternative approaches to solving two-loop (and higher) integrals have been developed, such as partial integration [ChTk 1981] and asymptotic expansions in external momenta [DaT 1992].

The algorithms implemented in xloops don't use Feynman parametrization but are instead based on parallel/orthogonal decomposition of the loop integration space, discussed in the next section. For reasons of consistency, the same method is also applied to the evaluation of one-loop integrals (which appear in the reduction of two-loop functions).

5.2 Parallel/Orthogonal Decomposition

The idea of parallel/orthogonal (PO) decomposition (see [Coll 1984, chapter 4]) is to increase the applicability of eq. (5.10) in the presence of external momenta in the integrand by splitting the integration space into the D_{\parallel} -dimensional *parallel* space spanned by the external momenta, and the $D_{\perp} = (D - D_{\parallel})$ -dimensional *orthogonal* space, the orthogonal complement of the parallel space. Since the external momenta are four-dimensional, the parallel space has integer dimension and can be chosen to be minimal by a suitable Lorentz transformation, with the minimum dimension depending on the number of linearly independent external momenta.

Splitting all loop momenta

$$\mathbf{l} = l^{\mu} = (l_0, \dots, l_{D_{\parallel}-1}, \mathbf{l}_{\perp}) \quad (5.11)$$

and external momenta

$$\mathbf{q} = q^{\mu} = (q_0, \dots, q_{D_{\parallel}-1}, \mathbf{0}) \quad (5.12)$$

into their parallel and orthogonal components, scalar products between the momenta can be written explicitly in terms of individual parallel components and orthogonal loop momentum vectors:

$$l^2 = l_0^2 - \dots - l_{D_{\parallel}-1}^2 - \mathbf{l}_{\perp}^2, \quad (5.13)$$

$$\mathbf{l} \cdot \mathbf{q} = l_0 q_0 - \dots - l_{D_{\parallel}-1} q_{D_{\parallel}-1}. \quad (5.14)$$

The integrand of a loop function is then spherically symmetric with respect to the (Euclidean) vector \mathbf{l}_\perp , and eq. (5.10) can be applied to it, yielding²

$$\int d^D l f(\mathbf{l}) = \frac{2\pi^{D_\perp/2}}{\Gamma(D_\perp/2)} \int d^{D_\parallel} \mathbf{l}_\parallel \int_0^\infty dl_\perp l_\perp^{D_\perp-1} f(\mathbf{l}) \quad (5.15)$$

This reduces the set of integration variables to the parallel components of the loop momenta, the absolute values of the orthogonal components, and (for diagrams with more than one loop) the angles between the loop momenta.

For example, in a two-point diagram there is only one independent external momentum q due to momentum conservation. Assuming that q is time-like ($q^2 > 0$) one can always choose a frame of reference in which

$$q^\mu = (q_0, \mathbf{0}), \quad (5.16)$$

corresponding to a one-dimensional parallel space. Any loop momentum l can then be written as

$$l^\mu = \frac{l_0 q_0}{q^2} q^\mu + l_\perp^\mu. \quad (5.17)$$

Likewise, for the three-point case with two external momenta q_1, q_2 we can choose a two-dimensional parallel space (assuming that q_1 is time-like) with

$$\begin{aligned} q_1^\mu &= (q_{10}, 0, \mathbf{0}), \\ q_2^\mu &= (q_{20}, q_{21}, \mathbf{0}). \end{aligned} \quad (5.18)$$

The loop momentum is then

$$l^\mu = \frac{l_0 q_{10}}{q_1^2} q_1^\mu + \frac{l_0 q_{20} - l_1 q_{21}}{q_2^2} q_2^\mu + l_\perp^\mu. \quad (5.19)$$

By inserting (5.17) or (5.19) into (5.4) it is easy to see that after performing the Lorentz decomposition of step 1 above, any one-loop tensor integral can be written as a linear combination of *PO-decomposed one-loop tensor integrals* of the form

$$T_{(n)t_1 \dots t_n}^{p_0 \dots p_{D_\parallel-1} p_\perp} = \int d^{D_\parallel} l \frac{l_0^{p_0} \dots l_{D_\parallel-1}^{p_{D_\parallel-1}} l_\perp^{p_\perp}}{\prod_i^n P_i^{t_i}}, \quad (5.20)$$

with non-negative integer powers p_j of the loop components. Here we also allow arbitrary positive integer powers t_i of the propagators P_i , which appear in the evaluation of two-loop functions and the tensor reduction of one-loop functions.

Note that (5.20) vanishes for odd p_\perp because the denominator of the integrand is symmetric in l_\perp .

² Equation (5.15) can be considered the definition of D -dimensional integration.

5.3 Tensor Reduction

All PO-decomposed tensor integrals (5.20) can be reduced to scalar integrals of the form

$$S_{(n)t_1\dots t_n} = \int d^D l \frac{1}{\prod_i^n P_i^{t_i}} \quad (5.21)$$

by expressing the loop momentum components in terms of the propagators in a way similar to step 2 of the conventional method.

5.3.1 One-Loop Two-Point Integrals

For the two-point tensor integral with one time-like external momentum q (one parallel dimension) one can choose the internal momenta so the integral takes the form

$$T_{(2)t_1 t_2}^{p_0 p_\perp} = \int d^D l \frac{l_0^{p_0} l_\perp^{p_\perp}}{P_1^{t_1} P_2^{t_2}} \quad (5.22)$$

with

$$\begin{aligned} P_1 &= (l_0 + q_0)^2 - l_\perp^2 - m_1^2 + i\rho, \\ P_2 &= l_0^2 - l_\perp^2 - m_2^2 + i\rho. \end{aligned} \quad (5.23)$$

In the general case where $q_0 \neq 0$ one can rewrite the numerator of the integrand in terms of the propagators by recursively substituting

$$\begin{aligned} l_0 &= \frac{1}{2q_0} (P_1 - P_2 - q_0^2 + m_1^2 - m_2^2), \\ l_\perp^2 &= l_0^2 - P_2 - m_2^2 + i\rho, \end{aligned} \quad (5.24)$$

converting the integral into a sum of the form

$$\begin{aligned} T_{(2)t_1 t_2}^{p_0 p_\perp} &= \sum_{i_1, i_2}^{p_0 + p_\perp} C_{i_1 i_2} \int d^D l P_1^{i_1 - t_1} P_2^{i_2 - t_2} \\ &= \sum_{i_1, i_2}^{p_0 + p_\perp} C_{i_1 i_2} \int d^D l P_1^{n_1} P_2^{n_2} \end{aligned} \quad (5.25)$$

with coefficients $C_{i_1 i_2}$ that can be calculated explicitly as functions of q , m_1 , and m_2 , without solving linear equations.

The sum in (5.25) contains three types of integrals, depending on the signs of $n_1 = i_1 - t_1$ and $n_2 = i_2 - t_2$:

1. $n_1 \geq 0$ and $n_2 \geq 0$:

$$\int d^D l P_1^{n_1} P_2^{n_2} = 0$$

because $\int d^D l (l^2)^\alpha = 0$ for any value of α in dimensional regularization, and because of the linearity of the D -dimensional integral.

2. $n_1 < 0$ and $n_2 < 0$:

$$\int d^D l \frac{1}{P_1^{-n_1} P_2^{-n_2}}$$

This is a scalar two-point function of the form (5.21).

3. $n_1 \geq 0$ and $n_2 < 0$, or $n_1 < 0$ and $n_2 \geq 0$:

$$\int d^D l \frac{P_1^{n_1}}{P_2^{-n_2}} \text{ or } \int d^D l \frac{P_2^{n_2}}{P_1^{-n_1}}$$

These are one-point functions which can be decomposed into integrals of the type

$$T_{(1)t}^p = \int d^D l \frac{(l^2)^{\frac{p}{2}}}{(l^2 - m^2 + i\rho)^t} \quad (5.26)$$

which are evaluated explicitly (they could in turn be reduced to scalar one-point functions but it is more efficient to calculate them directly).

In the case $q_0 = 0$, the propagators (5.23) take the form

$$\begin{aligned} P_1 &= l_0^2 - l_1^2 - m_1^2 + i\rho, \\ P_2 &= l_0^2 - l_1^2 - m_2^2 + i\rho. \end{aligned} \quad (5.27)$$

For $m_1 = m_2$ the propagators are equal and (5.22) takes the form of a one-point function with one parallel dimension $T_{(1)t_1+t_2}^{p_0 p_\perp}$.

For $m_1 \neq m_2$ one can perform a partial fraction decomposition in the masses to obtain

$$\begin{aligned} T_{(2)t_1 t_2}^{p_0 p_\perp} \Big|_{q_0=0} &= \frac{1}{(t_1 - 1)! d(m_1^2)^{t_1-1}} \left(\frac{1}{(m_1^2 - m_2^2)^{t_2}} \int d^D l \frac{l_0^{p_0} l_\perp^{p_\perp}}{P_1} \right) \\ &+ \frac{1}{(t_2 - 1)! d(m_2^2)^{t_2-1}} \left(\frac{1}{(m_2^2 - m_1^2)^{t_1}} \int d^D l \frac{l_0^{p_0} l_\perp^{p_\perp}}{P_2} \right) \end{aligned} \quad (5.28)$$

which again is a combination of one-point functions.

If the external momentum q is not time-like, it can no longer be brought into the form (5.16). It is, however, always possible to find a Lorentz transformation to a two-dimensional parallel space in which q takes the form

$$q^\mu = (q_0, q_1, \mathbf{0}). \quad (5.29)$$

As described in [BaDo 2002], the corresponding integral function

$$T_{(2)t_1 t_2}^{p_0 p_1 p_\perp} = \int d^D l \frac{l_0^{p_0} l_1^{p_1} l_\perp^{p_\perp}}{P_1^{t_1} P_2^{t_2}} \quad (5.30)$$

can be reduced to scalar functions by our method only for certain configurations of the internal momenta. It can, however, be calculated for all cases by performing suitable Lorentz transformations.

The two-point function with two parallel components (5.30) also appears during the reduction of three-point integrals.

5.3.2 One-Loop Three-Point Integrals

After choosing the specific frame of reference (5.18) and a two-dimensional parallel space, the one-loop three-point integral reads

$$T_{(3)t_1 t_2 t_3}^{p_0 p_1 p_\perp} = \int d^D l \frac{l_0^{p_0} l_1^{p_1} l_\perp^{p_\perp}}{P_1^{t_1} P_2^{t_2} P_3^{t_3}} \quad (5.31)$$

with

$$\begin{aligned} P_1 &= l_0^2 - l_1^2 - l_\perp^2 + 2l_0 q_{10} + q_{10}^2 - m_1^2 + i\rho, \\ P_2 &= l_0^2 - l_1^2 - l_\perp^2 + 2l_0 q_{20} - 2l_1 q_{21} + q_{20}^2 - q_{21}^2 - m_2^2 + i\rho, \\ P_3 &= l_0^2 - l_1^2 - l_\perp^2 - m_3^2 + i\rho. \end{aligned} \quad (5.32)$$

Applying the same method as in the two-point case, one finds the substitutions

$$\begin{aligned} l_0 &= \frac{1}{2q_{10}}(P_1 - P_3 - q_{10}^2 + m_1^2 - m_3^2), \\ l_1 &= \frac{1}{2q_{21}}(P_3 - P_2 + 2q_{20}l_0 - m_2^2 + m_3^2 + q_{20}^2 - q_{21}^2), \\ l_\perp^2 &= l_0^2 - l_1^2 - P_3 - m_3^2 + i\rho, \end{aligned} \quad (5.33)$$

which, in the general case $q_{10} \neq 0, q_{21} \neq 0$, converts the integral into a sum of the form

$$\begin{aligned} T_{(3)t_1 t_2 t_3}^{p_0 p_1 p_\perp} &= \sum_{i_1, i_2, i_3}^{p_0+p_1+p_\perp} C_{i_1 i_2 i_3} \int d^D l P_1^{i_1-t_1} P_2^{i_2-t_2} P_3^{i_3-t_3} \\ &= \sum_{i_1, i_2, i_3}^{p_0+p_1+p_\perp} C_{i_1 i_2 i_3} \int d^D l P_1^{n_1} P_2^{n_2} P_3^{n_3}. \end{aligned} \quad (5.34)$$

This time, there appear four types of integrals depending on the powers of the propagators in the integrands:

1. Integrals with entirely positive powers that vanish.
2. Scalar three-point integrals of the form

$$\int d^D l \frac{1}{P_1^{-n_1} P_2^{-n_2} P_3^{-n_3}}.$$

3. Two-point integrals of the form

$$\int d^D l \frac{P_A^{n_A}}{P_B^{-n_B} P_C^{-n_C}}.$$

4. One-point integrals of the form

$$\int d^D l \frac{P_A^{n_A} P_B^{n_B}}{P_C^{-n_C}}.$$

The resulting one- and two-point integrals (types 3 and 4) are in turn treated as before.

In the case $q_{10} = 0$, the propagators (5.32) become

$$\begin{aligned} P_1 &= l_0^2 - l_1^2 - l_\perp^2 - m_1^2 + i\rho, \\ P_2 &= l_0^2 - l_1^2 - l_\perp^2 + 2l_0q_{20} - 2l_1q_{21} + q_{20}^2 - q_{21}^2 - m_2^2 + i\rho, \\ P_3 &= l_0^2 - l_1^2 - l_\perp^2 - m_3^2 + i\rho. \end{aligned} \quad (5.35)$$

For $m_1 = m_3$ the first and third propagators are equal and (5.31) becomes a two-point function with two parallel dimensions $T_{(2)t_1+t_3, t_2}^{p_0 p_1 p_\perp}$.

For $m_1 \neq m_3$ one can again perform a partial fraction decomposition to reduce the integral to a sum of two-point functions:

$$T_{(3)t_1 t_2 t_3}^{p_0 p_1 p_\perp} \Big|_{q_{10}=0} = \prod_{i=1}^3 \left(\frac{1}{(t_i - 1)! d(m_i^2)^{t_i - 1}} \right) \left[\frac{1}{m_1^2 - m_3^2} \left(\int d^D l \frac{l_0^{p_0} l_1^{p_1} l_\perp^{p_\perp}}{P_1 P_2} - \int d^D l \frac{l_0^{p_0} l_1^{p_1} l_\perp^{p_\perp}}{P_2 P_3} \right) \right] \quad (5.36)$$

If $q_{21} = 0$, the external momenta are $q_1^\mu = (q_{10}, \mathbf{0})$, $q_2^\mu = (q_{20}, \mathbf{0})$ and the parallel space is thus only one-dimensional. Defining the new orthogonal loop momentum component as $l_\perp'^2 = l_1^2 + l_\perp^2$ the integral becomes

$$T_{(3)t_1 t_2 t_3}^{p_0, p_1 + p_\perp} = \int d^D l \frac{l_0^{p_0} l_\perp'^{p_1 + p_\perp}}{P_1^{t_1} P_2^{t_2} P_3^{t_3}} \quad (5.37)$$

with the propagators

$$\begin{aligned} P_1 &= (l_0 + q_{10})^2 - l_\perp'^2 - m_1^2 + i\rho, \\ P_2 &= (l_0 + q_{20})^2 - l_\perp'^2 - m_2^2 + i\rho, \\ P_3 &= l_0^2 - l_\perp'^2 - m_3^2 + i\rho. \end{aligned} \quad (5.38)$$

This integral can be reduced to a set of one-, two-, and scalar three-point functions with substitutions similar to those of the two-point case:

$$\begin{aligned} l_0 &= \frac{1}{2q_{10}} (P_1 - P_3 - q_{10}^2 + m_1^2 - m_3^2), \\ l_\perp'^2 &= l_0^2 - P_3 - m_3^2 + i\rho. \end{aligned} \quad (5.39)$$

With the exception of the two-point function with a two-dimensional parallel space (5.30), all other two- and three-point tensor integrals can be reduced to linear combinations of scalar functions for which explicit analytical results are well-known.

The same approach as outlined here for the one-loop integrals can also be applied to reduce two-loop functions to a set of basic (but not necessarily scalar) integrals, as described in [Do 2003, chapter 4].

5.4 Evaluation of the Basic One-Loop Integrals

The output of the one-loop tensor reduction algorithm of the previous section is an expression containing the following five basic integrals:

$$\begin{aligned}
T_{(1)t}^p &= \int d^D l \frac{(l^2)^{\frac{p}{2}}}{P^t}, \\
T_{(1)t}^{p_0 p_\perp} &= \int d^D l \frac{l_0^{p_0} l_\perp^{p_\perp}}{P^t}, \\
S_{(2)t_1 t_2} &= \int d^D l \frac{1}{P_1^{t_1} P_2^{t_2}}, \\
T_{(2)t_1 t_2}^{p_0 p_1 p_\perp} &= \int d^D l \frac{l_0^{p_0} l_1^{p_1} l_\perp^{p_\perp}}{P_1^{t_1} P_2^{t_2}}, \\
S_{(3)t_1 t_2 t_3} &= \int d^D l \frac{1}{P_1^{t_1} P_2^{t_2} P_3^{t_3}},
\end{aligned} \tag{5.40}$$

which are the tensor one-point functions with zero and one parallel dimension, the scalar two-point function with one parallel dimension, the tensor two-point function with two parallel dimensions, and the scalar three-point function, respectively.

`xloops` is able to perform the ε -expansion of these integrals by means of R functions [BFK 1994]. The one-point and the scalar two- and three-point functions have been reimplemented using GiNaC in [Baue 2000], while the two-point function with two parallel dimensions has been implemented in [Do 2003]. The calculation methods are based on [Brüç 1997].

The result of the expansion is a Laurent series (a GiNaC `pseries` object) with coefficients containing analytical functions like logarithms and dilogarithms.

5.5 The Hierarchy of One-Loop Integral Functions in `xloops`

The one-loop integral functions in `xloops` comprise of both C++ and symbolic GiNaC functions which are arranged in several “levels” (see fig. 5.1). They do not form a hierarchy by means of any inheritance relationships but by the higher level functions being expressible in terms of the lower ones.

The main difference between these functions is the form of the numerator of the integrand. We will explain them in the following sections, starting at the bottom of the hierarchy.

Appendix C gives the relationship of the `xloops` integral functions to the ones commonly used in the literature (A_0 , B_0 etc.).

5.5.1 The Scalar Functions `ScalarNPt()`

The scalar functions `Scalar1Pt()`, `Scalar2Pt()`, and `Scalar3Pt()` are symbolic functions which can appear in GiNaC expressions. They represent scalar integrals of the form (5.21) and are defined as follows:

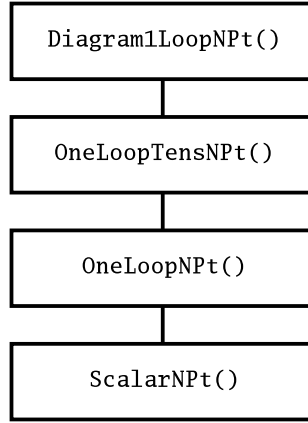
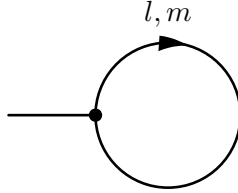


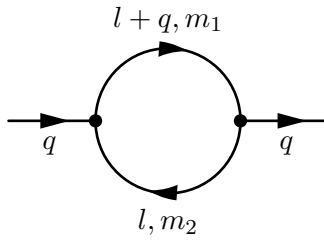
Figure 5.1: The hierarchy of one-loop integral functions in *xloops*. The functions in the upper groups can in general be represented in terms of those in the lower groups.

One-Point Function



$$\text{Scalar1Pt}(m, \rho, \varepsilon) := S_{(1)1} = \int d^D l \frac{1}{l^2 - m^2 + i\rho} \quad (5.41)$$

Two-Point Function

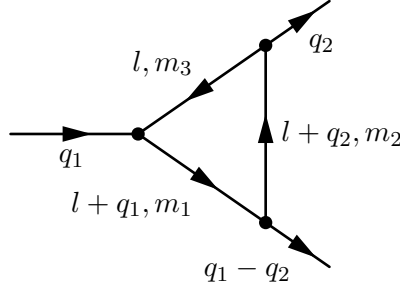


$$\begin{aligned} \text{Scalar2Pt}(q_0, m_1, m_2, t_1, t_2, \rho, \varepsilon) &:= S_{(2)t_1 t_2} \\ &= \int d^D l \frac{1}{((l + \mathbf{q})^2 - m_1^2 + i\rho)^{t_1} (l^2 - m_2^2 + i\rho)^{t_2}} \end{aligned} \quad (5.42)$$

with

$$\mathbf{q} = (q_0, \mathbf{0})$$

Three-Point Function



$$\begin{aligned} \text{Scalar3Pt}(q_{10}, q_{20}, q_{21}, m_1, m_2, m_3, t_1, t_2, t_3, \rho, \varepsilon) &:= S_{(3)t_1 t_2 t_3} \\ &= \int d^D l \frac{1}{((\mathbf{l} + \mathbf{q}_1)^2 - m_1^2 + i\rho)^{t_1} ((\mathbf{l} + \mathbf{q}_2)^2 - m_2^2 + i\rho)^{t_2} (\mathbf{l}^2 - m_3^2 + i\rho)^{t_3}} \end{aligned} \quad (5.43)$$

with

$$\begin{aligned} \mathbf{q}_1 &= (q_{10}, 0, \mathbf{0}), \\ \mathbf{q}_2 &= (q_{20}, q_{21}, \mathbf{0}) \end{aligned}$$

5.5.2 The PO-decomposed Tensor Functions `OneLoopNPt()`

The three functions `OneLoop1Pt()`, `OneLoop2Pt()`, and `OneLoop3Pt()` represent PO-decomposed tensor integrals of the form (5.20) in their respective minimal parallel spaces. The additional functions `SOneLoop1Pt()` and `S322Pt()` with a higher parallel space appear in the output of the tensor reduction algorithm described in section 5.3. These functions are also implemented as symbolic GiNaC functions.

The C++ function

```
ex tensor_reduction(const ex & e, const ex & eps)
```

can be called to explicitly reduce all tensor integral functions appearing in the given GiNaC expression `e` to scalar functions as far as possible (with the exception of one-point functions). This is mainly useful for testing and instructional purposes because the reduction is also performed automatically during the ε -expansion of the tensor functions.

One-Point Functions

$$\text{OneLoop1Pt}(p, m, t, \rho, \varepsilon) := T_{(1)t}^p = \int d^D l \frac{(\mathbf{l}^2)^{\frac{p}{2}}}{(\mathbf{l}^2 - m^2 + i\rho)^t} \quad (5.44)$$

$$\text{SOneLoop1Pt}(p_0, p_\perp, m, t, \rho, \varepsilon) := T_{(1)t}^{p_0 p_\perp} = \int d^D l \frac{l_0^{p_0} l_\perp^{p_\perp}}{(\mathbf{l}^2 - m^2 + i\rho)^t} \quad (5.45)$$

Two-Point Functions

$$\begin{aligned} \text{OneLoop2Pt}(p_0, p_\perp, q_0, m_1, m_2, t_1, t_2, \rho, \varepsilon) &:= T_{(2)t_1 t_2}^{p_0 p_\perp} \\ &= \int d^D l \frac{l_0^{p_0} l_\perp^{p_\perp}}{((\mathbf{l} + \mathbf{q})^2 - m_1^2 + i\rho)^{t_1} (\mathbf{l}^2 - m_2^2 + i\rho)^{t_2}} \end{aligned} \quad (5.46)$$

with

$$\mathbf{q} = (q_0, \mathbf{0}) \quad (5.47)$$

$$\begin{aligned} \text{S322Pt}(p_0, p_1, p_\perp, q_{10}, q_{20}, q_{21}, m_1, m_2, t_1, t_2, \rho, \varepsilon) &:= T_{(2)t_1 t_2}^{p_0 p_1 p_\perp} \\ &= \int d^D l \frac{l_0^{p_0} l_1^{p_1} l_\perp^{p_\perp}}{((\mathbf{l} + \mathbf{q}_1)^2 - m_1^2 + i\rho)^{t_1} ((\mathbf{l} + \mathbf{q}_2)^2 - m_2^2 + i\rho)^{t_2}} \end{aligned} \quad (5.48)$$

with

$$\begin{aligned} \mathbf{q}_1 &= (q_{10}, 0, \mathbf{0}), \\ \mathbf{q}_2 &= (q_{20}, q_{21}, \mathbf{0}) \end{aligned}$$

Three-Point Function

$$\begin{aligned} \text{OneLoop3Pt}(p_0, p_1, p_\perp, q_{10}, q_{20}, q_{21}, m_1, m_2, m_3, t_1, t_2, t_3, \rho, \varepsilon) &:= T_{(3)t_1 t_2 t_3}^{p_0 p_1 p_\perp} \\ &= \int d^D l \frac{l_0^{p_0} l_1^{p_1} l_\perp^{p_\perp}}{((\mathbf{l} + \mathbf{q}_1)^2 - m_1^2 + i\rho)^{t_1} ((\mathbf{l} + \mathbf{q}_2)^2 - m_2^2 + i\rho)^{t_2} (\mathbf{l}^2 - m_3^2 + i\rho)^{t_3}} \end{aligned} \quad (5.49)$$

with

$$\begin{aligned} \mathbf{q}_1 &= (q_{10}, 0, \mathbf{0}), \\ \mathbf{q}_2 &= (q_{20}, q_{21}, \mathbf{0}) \end{aligned}$$

5.5.3 The Lorentz Tensor Functions OneLoopTensNPt()

The C++ functions `OneLoopTens1Pt()`, `OneLoopTens2Pt()`, and `OneLoopTens3Pt()` represent *conventional* Lorentz tensor integrals (5.4) with loop momentum Lorentz indices μ_i . The external momenta are, however, still kept in a PO-decomposed form with explicit components.

One-Point Function

$$\text{OneLoopTens1Pt}(n, m, t, \rho, \varepsilon, C, \{\mu_1 \dots \mu_n\}) := T_{(1)t}^{\mu_1 \dots \mu_n} = \int d^D l \frac{l^{\mu_1} \dots l^{\mu_n}}{(\mathbf{l}^2 - m^2 + i\rho)^t} \quad (5.50)$$

Two-Point Function

$$\begin{aligned} \text{OneLoopTens2Pt}(n, q, m_1, m_2, t_1, t_2, \rho, \varepsilon, C, \{\mu_1 \dots \mu_n\}) &:= T_{(2)t_1 t_2}^{\mu_1 \dots \mu_n} \\ &= \int d^D l \frac{l^{\mu_1} \dots l^{\mu_n}}{((\mathbf{l} + \mathbf{q})^2 - m_1^2 + i\rho)^{t_1} (\mathbf{l}^2 - m_2^2 + i\rho)^{t_2}} \end{aligned} \quad (5.51)$$

Three-Point Function

$$\begin{aligned} \text{OneLoopTens3Pt}(n, q_1, q_2, q_{20}, q_{21}, m_1, m_2, t_1, t_2, t_3, \rho, \varepsilon, C, \{\mu_1 \dots \mu_n\}) &:= T_{(3)t_1 t_2 t_3}^{\mu_1 \dots \mu_n} \\ &= \int d^D l \frac{l^{\mu_1} \dots l^{\mu_n}}{((\mathbf{l} + \mathbf{q}_1)^2 - m_1^2 + i\rho)^{t_1} ((\mathbf{l} + \mathbf{q}_2)^2 - m_2^2 + i\rho)^{t_2} (\mathbf{l}^2 - m_3^2 + i\rho)^{t_3}} \end{aligned} \quad (5.52)$$

The Lorentz tensor functions return two items: The first is the Lorentz decomposition of the tensor as a GiNaC expression in a form similar to (5.5) with symbolic coefficients. The second is a list C of definitions for these coefficients in terms of the PO-decomposed tensor integral functions `OneLoopNPt()`. This list can be directly passed to the `subs()` method to replace the coefficients in the returned Lorentz decomposition by their actual values.

To calculate the coefficients, `xloops` first constructs the most general symmetric tensor from combinations of the metric and the external momenta. Then it extracts all possible individual components of the tensor in the PO-decomposed space-time. The resulting linear system of equations can be solved iteratively because the external momenta have been chosen so their higher components vanish.

5.5.4 The Loop Diagram Functions `Diagram1LoopNPt()`

The C++ functions `Diagram1Loop2Pt()` and `Diagram1Loop3Pt()` are an attempt at a high-level frontend for the `xloops` one-loop library and an interface between it and the graph generator. These functions calculate a one-loop truncated Green function of the form (5.1) with an arbitrary numerator N that may contain scalar terms, indexed loop momenta (l^μ), slashed loop momenta (\not{l}), and scalar products of loop and/or external momenta (l^2 , $l \cdot q_i$), calling the corresponding `OneLoopTensNPt()` functions for each case. They also post-process the result to bring it into an easily readable form.

The numerator N will eventually be assembled automatically via insertion of the Feynman rules in the output of the `xloops` graph generator. Currently this has to be done by hand, however.

Two-Point Function

$$\begin{aligned} & \text{Diagram1Loop2Pt}(N, q, m_1, m_2, l) \\ &= \mu^{4-D} \int \frac{d^D l}{(2\pi)^D} \frac{N}{((\mathbf{l} + \mathbf{q})^2 - m_1^2 + i\rho)(\mathbf{l}^2 - m_2^2 + i\rho)} \end{aligned} \quad (5.53)$$

Three-Point Function

$$\begin{aligned} & \text{Diagram1Loop3Pt}(N, q_1, q_2, m_1, m_2, m_3, l) \\ &= \mu^{4-D} \int \frac{d^D l}{(2\pi)^D} \frac{N}{((\mathbf{l} + \mathbf{q}_1)^2 - m_1^2 + i\rho)((\mathbf{l} + \mathbf{q}_2)^2 - m_2^2 + i\rho)(\mathbf{l}^2 - m_3^2 + i\rho)} \end{aligned} \quad (5.54)$$

To keep intermediate expressions compact, the loop diagram functions store them as expression pairs (two-element GiNaC lists) $\{c, n(l)\} := c \cdot n(l)$ with c being independent of the loop momentum l . Results are also returned in this format. For example, the expression $ie\not{l} - 4iem\mathbb{1}$ may be represented as the sum $\{ie, \not{l}\} + \{-4iem\mathbb{1}, 1\}$ or as the single pair $\{ie, \not{l} - 4m\mathbb{1}\}$.

`Diagram1Loop2Pt()` and `Diagram1Loop3Pt()` perform the following steps:

1. The numerator N is converted to a $\{c, n\}$ pair. The overall factor $\frac{\mu^{4-D}}{(2\pi)^D}$ is also multiplied into c .
2. `simplify_indexed()` (see section 3.3) is run over n to perform Lorentz contractions and to bring all scalar products into a canonical form.
3. n , which in general is a sum, is split into terms, with each term (plus the common factor c) being converted to an individual $\{c, n\}$ pair.
4. All occurrences of the loop momentum l in each n are rewritten with explicit (temporary) indices (for example $l \rightarrow l^\mu \gamma_\mu$, $l^2 \rightarrow l^\mu l^\nu \eta_{\mu\nu}$), to bring them into a form suitable for calling the tensor integral functions.
5. The appropriate `OneLoopTensNPt()` or `ScalarNPt()` function is called for each term.
6. Factors of $\frac{1}{D_\perp}$ appearing in the results are expanded up to order $\mathcal{O}(\varepsilon^0)$.
7. `simplify_indexed()` is run again to contract indices of the external momenta and the metric tensor.
8. The terms are added up to a single $\{c, n\}$ pair. n is collected in terms of the `xloops` scalar and tensor integral functions.
9. The Lorentz decomposition and contractions may result in factors containing $D = 4 - 2\varepsilon$ in the numerator which cancel poles in the integral functions. Such cancellations are detected and calculated explicitly.
10. Any remaining common factor in the result is again divided out and moved to c .
11. The coefficients of the loop integral functions in n are brought into a normalized rational form by invoking `normal()`.
12. The final $\{c, n\}$ pair is returned as the result of the function.

We will give an example for the use and the internal operation of the loop diagram functions in the next section.

The `Diagram1LoopNPt()` functions are not yet completely general as they still assume the external momenta to be in a PO-decomposed form with a minimal parallel space. This especially limits their usefulness for three-point calculations. Expressing the results in terms of invariants will be the subject of future work.

5.6 Sample Calculations

We conclude this chapter with two examples for one-loop calculations which, except for the generation of the contributing graphs and the insertion of the Feynman rules, can be performed in a fully automated manner by `xloops`.

For the used Feynman rules and other conventions like the definition of the proper self-energy we follow [BDJ 2001, appendix A].

5.6.1 QED Electron Self-Energy

The lowest contribution to the electron self-energy $\Sigma(q)$ in QED results from the diagram

$$\Sigma(q) = -i \left[\text{Diagram} \right] \quad (5.55)$$

and can be decomposed into a vector and a scalar part

$$\Sigma(q) = \Sigma_V(q^2) \not{q} + \Sigma_S(q^2) m \mathbb{1}. \quad (5.56)$$

Both parts are computed in a straightforward way by the following program fragment:

```

1 // definitions of momenta, electron charge and mass, and indices
2 symbol l("l"), q("q"), e("e"), m("m");
3 varidx mu(symbol("mu"), D), nu(symbol("nu"), D);
4
5 // construct numerator of integral from Feynman rules
6 ex N = -I;
7 N *= I * e * dirac_gamma(nu.toggle_variance());
8 N *= I * (dirac_slash(l, D) + dirac_slash(q, 4) + dirac_ONE()*m);
9 N *= I * e * dirac_gamma(mu.toggle_variance());
10 N *= -I * lorentz_g(mu, nu);
11
12 // call two-point diagram function
13 ex dia = Diagram1Loop2Pt(N, q, m, 0, 1);
14
15 // extract parts
16 ex Sigma_V = dia[0] * dia[1].expand().coeff(dirac_slash(q, 4));
17 ex Sigma_S = dia[0] * dia[1].expand().coeff(dirac_ONE()).coeff(m);

```

which yields the well-known result (cf. [BDJ 2001, eq. (2.5.97)])

$$\begin{aligned} \Sigma_V &= -\frac{e^2}{16\pi^4} (2\pi\mu)^{2\varepsilon} \left(2i \text{Scalar2Pt}(q, m, 0) + 2i \frac{\text{OneLoop2Pt}(1, 0, q, m, 0)}{\sqrt{q^2}} + \pi^2 \right) \\ &= -\frac{\alpha}{4\pi} \left(\frac{A_0(m^2) - (q^2 + m^2)B_0(q^2; m, 0)}{q^2} + 1 \right), \\ \Sigma_S &= \frac{e^2}{16\pi^4} (2\pi\mu)^{2\varepsilon} (4i \text{Scalar2Pt}(q, m, 0) + 2\pi^2) \\ &= -\frac{\alpha}{4\pi} (4B_0(q^2; m, 0) - 2), \end{aligned}$$

in the limit $\varepsilon \rightarrow 0$.

The `Diagram1Loop2Pt()` function first (step 1) separates the constant factor $c = \frac{e^2}{16\pi^4}(2\pi\mu)^{2\varepsilon}$ from the numerator which then reads

$$(\gamma_\nu(i\not{l} + i\not{q} + im\not{1})\gamma_\mu)\eta^{\mu\nu}.$$

Performing the index simplification and expansion (steps 2 and 3) results in the five terms

$$imD\not{1} + 2i\not{q} - iD\not{q} - iD\not{l} + 2i\not{l}.$$

The first three terms correspond to scalar integrals which are directly converted to `Scalar2Pt()` functions. The two remaining terms are integrals proportional to \not{l} for which invoking `OneLoopTens2Pt()` (steps 4–7) returns the result $\frac{\not{q}}{\sqrt{q^2}}\text{OneLoop2Pt}(1, 0, q, m, 0)$.

The factor D arises from the contractions $\gamma^\mu\not{l}\gamma_\mu$ and $\gamma^\mu\not{q}\gamma_\mu$. After recombining the five individual terms, the total $\mathcal{O}(\varepsilon^1)$ term is calculated as

$$\begin{aligned} \varepsilon \left(2i\frac{\not{q}}{\sqrt{q^2}}\text{OneLoop2Pt}(1, 0, q, m, 0) + 2i(\not{q} - m\not{1})\text{Scalar2Pt}(q, m, 0) \right) \\ = \pi^2(2m\not{1} - \not{q}) + \mathcal{O}(\varepsilon^1) \end{aligned}$$

by series expansion of the loop integral functions up to order $\mathcal{O}(\varepsilon^{-1})$ (step 9). Inserting this and omitting higher-order terms gives the final result

$$\begin{aligned} \Sigma = \frac{e^2}{16\pi^4}(2\pi\mu)^{2\varepsilon} \left(2i(2m\not{1} - \not{q})\text{Scalar2Pt}(q, m, 0) \right. \\ \left. - 2i\frac{\not{q}}{\sqrt{q^2}}\text{OneLoop2Pt}(1, 0, q, m, 0) + \pi^2(2m\not{1} - \not{q}) \right) \end{aligned}$$

from which the form factors can be directly extracted as shown in the program above.

5.6.2 Non-Diagonal Contributions to the Quark Self-Energy

Due to mixing, the proper self-energy of quarks contains terms which are non-diagonal in a basis of mass eigenstates (physical fields), and which arise from diagrams containing virtual charged bosons [DeSa 1990].

In the Standard Model, there are two contributions, one from the W boson exchange

$$\Sigma_{ij}^{(1)} = -i \left[\begin{array}{c} \text{Diagram: A quark line with momentum } q \text{ enters from the left at vertex } \mu \text{ and exits at vertex } \nu. \text{ A loop is formed by a quark line with momentum } k+q \text{ and a } W(k) \text{ boson line.} \\ \text{The quark line in the loop is labeled } u_l(k+q) \text{ and the } W \text{ boson line is labeled } W(k). \end{array} \right], \quad (5.57)$$

and one from the charged Higgs exchange

$$\Sigma_{ij}^{(2)} = -i \left[\begin{array}{c} \text{---} d_i(q) \text{---} \bullet \text{---} \text{---} u_l(k+q) \text{---} \bullet \text{---} \text{---} d_j(q) \text{---} \\ \text{---} \phi(k) \text{---} \end{array} \right], \quad (5.58)$$

where d_i and u_l indicate down- and up-type quark fields with flavor indices i and l , respectively.

The total non-diagonal self-energy Σ'_{ij} resolves into left-handed, right-handed, and scalar parts:³

$$\Sigma'_{ij} = \Sigma_{ij}^{(1)} + \Sigma_{ij}^{(2)} = \Sigma_{ij}^L \not{q} \gamma_L + \Sigma_{ij}^R \not{q} \gamma_R + \Sigma_{ij}^S (m_j \gamma_L + m_i \gamma_R)$$

These parts can be computed with a program similar to the one given above for the electron self-energy. In an R_ξ gauge with an unspecified gauge parameter ξ , the result is

$$\begin{aligned} \Sigma_{ij}^L &= \frac{ie^2}{32\pi^4 s_w^2} V_{jl}^+ V_{li} (2\pi\mu)^{2\epsilon} \cdot \left[i\pi^2 + \frac{1}{\sqrt{q^2}} \left(2 \text{OneLoop2Pt}(1, 0, q, M_W, m_l) + \frac{m_l^2}{M_W^2} \text{OneLoop2Pt}(1, 0, q, \sqrt{\xi} M_W, m_l) \right) \right. \\ &\quad + \frac{1}{M_W^2} \left(- \text{OneLoop2Pt}(2, 0, q, m_l, M_W) + \text{OneLoop2Pt}(2, 0, q, m_l, \sqrt{\xi} M_W) \right. \\ &\quad \quad - \text{OneLoop2Pt}(0, 2, q, m_l, M_W) + \text{OneLoop2Pt}(0, 2, q, m_l, \sqrt{\xi} M_W) \\ &\quad \quad + \frac{1}{\sqrt{q^2}} \text{OneLoop2Pt}(1, 2, q, m_l, M_W) - \frac{1}{\sqrt{q^2}} \text{OneLoop2Pt}(1, 2, q, m_l, \sqrt{\xi} M_W) \\ &\quad \quad \left. \left. - \frac{1}{\sqrt{q^2}} \text{OneLoop2Pt}(3, 0, q, m_l, M_W) + \frac{1}{\sqrt{q^2}} \text{OneLoop2Pt}(3, 0, q, m_l, \sqrt{\xi} M_W) \right) \right], \\ \Sigma_{ij}^R &= \frac{ie^2}{32\pi^4 s_w^2} V_{jl}^+ V_{li} (2\pi\mu)^{2\epsilon} \frac{m_i m_j}{M_W^2} \frac{1}{\sqrt{q^2}} \text{OneLoop2Pt}(1, 0, q, \sqrt{\xi} M_W, m_l), \\ \Sigma_{ij}^S &= \frac{ie^2}{32\pi^4 s_w^2} V_{jl}^+ V_{li} (2\pi\mu)^{2\epsilon} \frac{m_l^2}{M_W^2} \text{Scalar2Pt}(q, m_l, \sqrt{\xi} M_W), \end{aligned} \quad (5.59)$$

with s_w being the sine of the Weinberg angle, V_{ij} the quark mixing matrix, and V_{ij}^+ its adjoint.

³ The W contribution only has a left-handed part.

In the Feynman gauge ($\xi = 1$) this simplifies to

$$\begin{aligned}
 \Sigma_{ij}^L &= \frac{ie^2}{32\pi^4 s_w^2} V_{jl}^+ V_{li} (2\pi\mu)^{2\varepsilon} \cdot \left[i\pi^2 + \left(2 + \frac{m_l^2}{M_W^2} \right) \frac{1}{\sqrt{q^2}} \text{OneLoop2Pt}(1, 0, q, M_W, m_l) \right] \\
 &= -\frac{\alpha}{8\pi s_w^2} V_{jl}^+ V_{li} \left[1 + \left(2 + \frac{m_l^2}{M_W^2} \right) B_1(q^2; m_l, M_W) \right], \\
 \Sigma_{ij}^R &= \frac{ie^2}{32\pi^4 s_w^2} V_{jl}^+ V_{li} (2\pi\mu)^{2\varepsilon} \frac{m_i m_j}{M_W^2} \frac{1}{\sqrt{q^2}} \text{OneLoop2Pt}(1, 0, q, M_W, m_l) \quad (5.60) \\
 &= -\frac{\alpha}{8\pi s_w^2} V_{jl}^+ V_{li} \frac{m_i m_j}{M_W^2} B_1(q^2; m_l, M_W), \\
 \Sigma_{ij}^S &= \frac{ie^2}{32\pi^4 s_w^2} V_{jl}^+ V_{li} (2\pi\mu)^{2\varepsilon} \frac{m_l^2}{M_W^2} \text{Scalar2Pt}(q, m_l, M_W) \\
 &= -\frac{\alpha}{8\pi s_w^2} V_{jl}^+ V_{li} \frac{m_l^2}{M_W^2} B_0(q^2; m_l, M_W),
 \end{aligned}$$

in accordance with the analytical results of [DeSa 1990, eq. (3.8)] for a single intermediate quark flavor l . Explicit expressions for the divergent and finite parts of the result can be obtained by invoking `series()` and extracting the coefficients of ε^{-1} and ε^0 .

These three functions and their derivatives enter into the calculation of the wave-function renormalization constants of the quark fields (for non-vanishing neutrino masses this extends to all fermions).

6 Conclusions

Help, Mr. Wizard!

Tooter Turtle

The first public release of GiNaC in 1999 filled the demand for an extensible symbolic manipulation engine that was both available in source code form, as well as comparable in performance to established proprietary systems that often bind the user by highly restrictive licenses. Since that time, a number of other programs and libraries related to computer algebra such as AXIOM, NTL, PARI, and SINGULAR have been (re-)released under the Open Source model.¹

The implementation of GiNaC as a C++ library certainly sets it apart from other algebraic systems. Our results prove that a highly efficient symbolic manipulator can be designed in an object-oriented fashion, and that having a very fine granularity of objects is also feasible. The loss in raw speed due to this approach is clearly made up for by the flexibility gained on the user side. To extend the library one can simply add one's own customized classes to it. In traditional systems, the user first has to work out a suitable data representation in terms of the types and structures offered by the system.

In earlier versions, the design of GiNaC was not as consistent as it could or should have been. Its original conception as a "MAPLE emulator" led to some compromises such as the lack of iterators that have been largely amended as a part of this work. There are more opportunities left for additional future design adjustments. For example, GiNaC still uses lists in some places where vectors or sets would be more appropriate. Ideas for possible performance and feature enhancements of GiNaC have been discussed in chapter 4.

The `xloops`-related parts of this work consist of a new implementation, based on GiNaC, of functions that already existed in the original `xloops` as well as the addition of supplementary functions such as `Diagram1LoopMPt()` which belong to the interface between the library of integral functions and the graph generator. There are also many new modules for `xloops` under development or nearing completion. These include the mentioned graph generator, routines for a set of two-loop three-point topologies, and the one-loop four-point functions which are presently undergoing testing.

Higher-order loop integrals as well as the complete automatization of the entire computation chain from the description of processes up to the calculation of cross-sections will be future challenges. One option would be to delegate some of these tasks by interoperation with other existing program packages such as COMPHEP.

Eventually, the problem of an appropriate user interface for `xloops` will have to be tackled. While controlling the system from C++ offers the maximum amount of power and flexibility, it may be too cumbersome for the casual user. Another major problem is the

¹ According to personal communication with Jos Vermaseren, this is also planned for FORM.

comprehensible presentation of results, since the symbolic expressions returned by the current xloops are often far from being in the most “simple” (or intuitive) form. This is, however, an issue that GiNaC shares with every other computerized algebra system.

A Notation

To document the relationships between classes and objects we use a diagrammatic representation based on a simplified version of the Unified Modeling Language (UML) [Rati 1997] (see also appendix B of [GHJV 1995]).

A.1 Class Diagrams

A *class diagram* (fig. A.1) illustrates the relationships between classes. Each class is represented by a box with the class name at the top. Below the name, the class attributes and operations (if any) are listed. As in C++, the types of attributes, operations, and parameters are put before their respective identifier. Type names and method parameters are usually omitted if they are not relevant. Names of abstract classes or operations are set in an oblique typeface. Different kinds of arrows indicate inheritance, aggregation, and association between classes. Dog-eared boxes contain annotations on the implementation of operations in plain text or pseudocode.

A.2 Object Diagrams

An *object diagram* (fig. A.2) shows instances of classes in memory. Each rounded box represents one object, with its class name at the top and the values of its attributes below. Arrows point to aggregated or referenced objects.

A.3 Interaction Diagrams

A sequence of events such as object construction/deletion and method invocations are recorded in an *interaction diagram* (fig. A.3). To every participating object there belongs one *lifeline* on which time flows from top to bottom. A dashed lifeline indicates that the object does not yet or no longer exist at that point. The scope of a request is represented by a vertical box, and solid arrows show the invocation of methods. A request for the construction of an object is indicated by a dashed arrow.

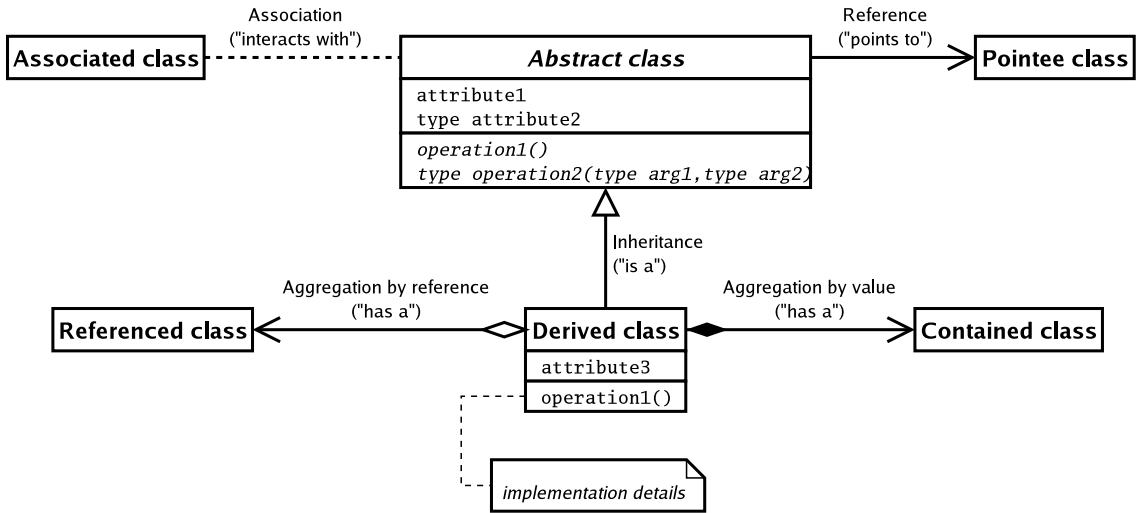


Figure A.1: Class diagram

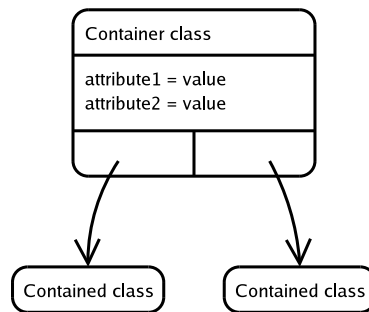


Figure A.2: Object diagram

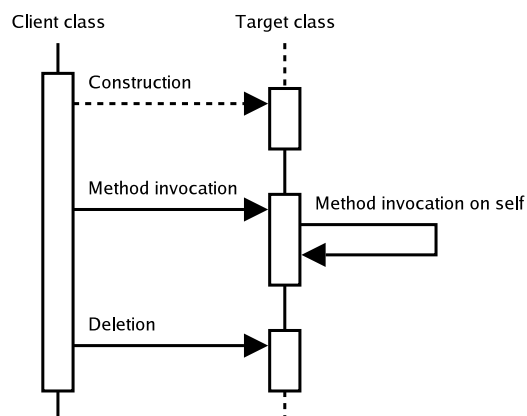


Figure A.3: Interaction diagram

B Simplification Rules

B.1 Predefined Tensors

The following simplification rules are implemented by the predefined tensor classes listed in section 3.2.2. Only rules specific to the respective tensors are given. Generic transformation rules implemented by GiNaC such as the canonicalization of summation indices (see section 3.3) are omitted.

In this section, we use small Roman letters to denote arbitrary indices (class `idx`), Greek letters for indices with variance (class `varidx`), and capital Roman letters for spinor indices (class `spinidx`). Values in square brackets `[]` indicate index dimensions.

B.1.1 The Kronecker delta δ_{ij} (`tensdelta`)

The Kronecker delta δ_{ij} is defined by

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \quad (\text{B.1})$$

Rules implemented in `tensdelta::eval_indexed()`:

- $\delta_{i[D_i]j[D_j]} \rightarrow \delta_{i[D]j[D]}$ with $D = \min(D_i, D_j)$ (δ_{ij} is made square)
- $\delta_{i[D]i[D]} \rightarrow D$ (trace of identity matrix)
- $i, j \in \mathbb{Z} \Rightarrow \delta_{ij} \rightarrow \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$ (explicit components)

Rules implemented in `tensdelta::contract_with()`:

- $\delta_{ij[D_\delta]} X_{j[D_X]} \rightarrow X_{i[D]}$ with $D = \min(D_\delta, D_X)$

B.1.2 The general metric tensor $g_{\mu\nu}$ (`tensmetric`)

The `tensmetric` class represents a symmetric, regular metric tensor that can be used to raise and lower tensor indices.

Rules implemented in `tensmetric::eval_indexed()`:

- $g_{\mu[D_\mu]\nu[D_\nu]} \rightarrow g_{\mu[D]\nu[D]}$ with $D = \min(D_\mu, D_\nu)$ ($g_{\mu\nu}$ is made square)

- $g^\mu{}_\nu \rightarrow \delta^\mu{}_\nu$, $g_\mu{}^\nu \rightarrow \delta_\mu{}^\nu$

Rules implemented in `tensmetric::contract_with()`:

- $g_{\mu\nu[D_g]}X^{\nu[D_X]} \rightarrow X_{\mu[D]}$ with $D = \min(D_g, D_X)$, except when $X \equiv \delta$, so $g_{\mu\nu}\delta^\nu{}_\rho$ produces $g_{\mu\rho}$, not $\delta_{\mu\rho}$

B.1.3 The Minkowski metric tensor $\eta_{\mu\nu}$ (`minkmetric`)

The Minkowski metric tensor inherits the simplification rules of the `tensmetric` class, plus the following matrix representations:

$$\eta_{\mu\nu} = \begin{pmatrix} 1 & 0 & 0 & \cdots \\ 0 & -1 & 0 & \\ 0 & 0 & -1 & \\ \vdots & & & \ddots \end{pmatrix} \quad (\text{negative signature}) \quad (\text{B.2})$$

$$\eta_{\mu\nu} = \begin{pmatrix} -1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \\ 0 & 0 & 1 & \\ \vdots & & & \ddots \end{pmatrix} \quad (\text{positive signature}) \quad (\text{B.3})$$

which are inserted when both indices have nonnegative integer values.

B.1.4 The spinor metric tensor ϵ_{AB} (`spinmetric`)

The antisymmetric spinor metric tensor ϵ_{AB} defines a Lorentz-invariant spinor product in the Weyl-van-der-Waerden formalism [Ditt 1998] and can be used to raise and lower spinor indices. Its matrix representation is

$$\epsilon_{AB} = \epsilon_{\dot{A}\dot{B}} = \epsilon^{AB} = \epsilon^{\dot{A}\dot{B}} = i\sigma^2 = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}. \quad (\text{B.4})$$

Rules implemented in `spinmetric::eval_indexed()`:

- $\epsilon_A{}^A \rightarrow 0$, $\epsilon^A{}_A \rightarrow 0$ (contractions vanish)
- $A, B \in \{0, 1\} \Rightarrow \epsilon_{AB} \rightarrow \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}_{AB}$ (explicit components)

Rules implemented in `spinmetric::contract_with()`:

- $\epsilon_{AB}\epsilon^{AC} \rightarrow \delta_B{}^C$ (other contractions follow from the antisymmetry of ϵ)
- $\epsilon^{AB}X_B \rightarrow X^A$, $\epsilon_{AB}X^B \rightarrow -X_A$, except when $X \equiv \delta$

B.1.5 The Levi-Civita tensor ϵ (tensepsilon)

The tensor $\epsilon_{ijk\dots}$ can be defined by

$$\epsilon_{\pi(0)\pi(1)\pi(2)\dots} = \text{sgn } \pi \quad (\text{B.5})$$

for any permutation π of index values. Components with one or more equal index values are zero. The number of indices must be equal to the index dimension (the cardinality of the index set) which implies that the index dimension must be a positive integer.

In a Minkowski space, one also has to observe that

$$\epsilon^0{}_{123} = \eta^{0\mu} \epsilon_{\mu 123} = \eta^{00} \epsilon_{0123} = -\epsilon_{0123} \quad (\text{B.6})$$

for $\eta^{\mu\nu} = \text{diag}(-1, 1, 1, 1)$ (for $\eta^{\mu\nu} = \text{diag}(1, -1, -1, -1)$ the sign change appears in the other three components).

Rules implemented in `tensepsilon::eval_indexed()`:

- $\epsilon^\mu{}_{\mu\dots} \rightarrow 0$ (contractions vanish)
- If all indices are nonnegative integers, the tensor is evaluated according to (B.5) and (B.6).

Rules implemented in `tensepsilon::contract_with()`:

- Contractions between two ϵ tensors are evaluated as a determinant, for example in three dimensions

$$\epsilon_{ijk} \epsilon_{lmn} = \begin{vmatrix} \delta_{il} & \delta_{jl} & \delta_{kl} \\ \delta_{im} & \delta_{jm} & \delta_{km} \\ \delta_{in} & \delta_{jn} & \delta_{kn} \end{vmatrix}, \quad (\text{B.7})$$

which can be generalized to arbitrary dimensions. In the case of indices with variance, the δ_{ij} are replaced by the appropriate metric tensors $g_{\mu\nu}$ or $\eta_{\mu\nu}$. In a Minkowski space, there also appears an additional minus sign because of (B.6).

B.2 Dirac Algebra

The simplification rules implemented by the Dirac algebra objects $\mathbb{1}$, γ^μ , γ^5 , $\gamma^L = \frac{1}{2}(1 - \gamma^5)$, $\gamma^R = \frac{1}{2}(1 + \gamma^5)$, and $\not{\psi}$ described in section 3.4.1 fall into two categories: canonicalization of Dirac strings and contractions of Dirac matrices.

The following set of rules is applied in `clifford::eval_ncmul()` to canonicalize Dirac strings:

1. All occurrences of $\mathbb{1}$ are removed.
2. Occurrences of γ^5 , γ^L , and γ^R are moved to the beginning of the string using the (anti)commutation relations

$$\begin{aligned} \gamma^\mu \gamma^5 &= -\gamma^5 \gamma^\mu, & \gamma^\mu \gamma^L &= \gamma^R \gamma^\mu, & \gamma^\mu \gamma^R &= \gamma^L \gamma^\mu, \\ \not{\psi} \gamma^5 &= -\gamma^5 \not{\psi}, & \not{\psi} \gamma^L &= \gamma^R \not{\psi}, & \not{\psi} \gamma^R &= \gamma^L \not{\psi}, \\ \gamma^L \gamma^5 &= \gamma^5 \gamma^L, & \gamma^R \gamma^5 &= \gamma^5 \gamma^R. \end{aligned}$$

3. Squares of Dirac matrices are removed using

$$\begin{aligned} \gamma^5 \gamma^5 &= \mathbb{1}, & \gamma^L \gamma^L &= \gamma^L, & \gamma^R \gamma^R &= \gamma^R, \\ \gamma^\mu \gamma^\mu &= \eta^{\mu\mu} \mathbb{1}, & \not{p} \not{p} &= p^\mu p_\mu \mathbb{1}. \end{aligned}$$

4. Remaining products of γ^5 , γ^L , and γ^R are simplified using

$$\gamma^5 \gamma^L = -\gamma^L, \quad \gamma^5 \gamma^R = \gamma^R, \quad \gamma^L \gamma^R = \gamma^R \gamma^L = 0.$$

The string now has either exactly one instance of γ^5 , γ^L , or γ^R as its first element, or none at all.

5. Commutative factors like $\eta^{\mu\mu}$, p^μ , and accumulated minus signs are pulled out of the string.

6. If no Dirac matrices are left in the resulting string it is replaced by a single $\mathbb{1}$.

If necessary, the rules are applied repeatedly until a stable state is reached.

The only Dirac object implementing contractions is γ^μ as it is the only one with a proper index. The following rules are implemented in `diracgamma::contract_with()`:

- $\gamma^\mu \gamma_\mu \rightarrow \dim(\mu) \mathbb{1}$
- $\gamma^\mu \gamma^\alpha \gamma_\mu \rightarrow (2 - \dim(\mu)) \gamma^\alpha$
- In four dimensions: $\gamma^\mu S_{\text{odd}} \gamma_\mu \rightarrow -2S_{\text{odd}}^R$ and $\gamma^\mu S_{\text{odd}} \gamma^\alpha \gamma_\mu \rightarrow 2\gamma^\alpha S_{\text{odd}} + 2S_{\text{odd}}^R \gamma^\alpha$ for any string S_{odd} containing an odd number of Dirac matrices. S_{odd}^R is the reversed string.
- $\gamma^\mu S \gamma^\alpha \gamma_\mu \rightarrow 2\gamma^\alpha S - \gamma^\mu S \gamma_\mu \gamma^\alpha$ for any string S . The remaining contraction $\gamma^\mu S \gamma_\mu$ is then evaluated recursively.
- $\gamma^\mu p_\mu \rightarrow \not{p}$ if p is a symbol

Contractions between Dirac matrices are only carried out when their representation labels are equal.

The `eval_indexed()` method is not used by any of the Dirac classes.

B.3 Color Algebra

The simplification rules for the color algebra objects $\mathbb{1}_C$, T_a , f_{abc} , and d_{abc} described in section 3.4.2 refer almost exclusively to contractions of color objects. The rules for the Kronecker delta δ_{ab} have already been listed in appendix B.1.1.

The only simplification performed on color strings in `color::eval_ncmul()` is to remove instances of $\mathbb{1}_C$ unless the string only contains one single $\mathbb{1}_C$.

The rules implemented for the generators T_a of the fundamental representation of $SU(3)$ in `su3t::contract_with()` are:

- $T_a T_a \rightarrow \frac{4}{3} \mathbb{1}_C$ ($= C_F \mathbb{1}_C$, the Casimir operator of $SU(3)$ in this representation)
- $T_a T_b T_a \rightarrow -\frac{1}{6} T_b$
- $T_a S T_a \rightarrow \frac{1}{2} \text{Tr}[S] \mathbb{1}_C - \frac{1}{6} S$ for any string S with two or more elements

All contractions are only carried out when the representation labels of the objects are equal.

The indices of the symmetric and antisymmetric structure constants d_{abc} and f_{abc} are automatically brought into a canonical order in `indexed::eval()` according to the symmetry properties attached to them (see section 3.1). Additionally, both objects evaluate to numbers when all indices are numeric, like $f_{246} \rightarrow \frac{1}{2}$, $d_{338} \rightarrow \frac{1}{\sqrt{3}}$. This is implemented in `su3d::eval_indexed()` and `su3f::eval_indexed()`.

Finally, the following contraction rules are applied in `su3d::contract_with()` and `su3f::contract_with()`:

- $d_{abc} d_{abc} \rightarrow \frac{40}{3}$
- $d_{aij} d_{bij} \rightarrow \frac{5}{3} \delta_{ab}$
- $d_{abc} T_b T_c \rightarrow \frac{5}{6} T_a$
- $f_{abc} f_{abc} \rightarrow 24$
- $f_{aij} f_{bij} \rightarrow 3 \delta_{ab}$ ($= C_A \delta_{ab}$, the Casimir operator in the adjoint representation of $SU(3)$)
- $f_{abc} T_b T_c \rightarrow \frac{3i}{2} T_a$

All rules also apply for different orders of the indices. Contractions between f_{abc} and d_{abc} that vanish for symmetry reasons are handled by `simplify_indexed()`.

C Relation of xloops Functions to Standard One-Loop Integrals

The functions implemented in the xloops library (cf. section 5.5) are related to, but not identical to the standard one-loop integral functions commonly found in the literature (see, for example [BDJ 2001]). Aside from the PO decomposition of the external momenta and factors like $i\pi^2$, the momenta along the internal propagators are also assigned in a different way.

In this appendix, we will illustrate the differences between the two definitions considering some exemplary scalar and tensor functions. For simplicity, higher powers of propagators and the causality term $i\rho$ are omitted.

One-Point Functions

$$\begin{aligned}
 A_0(m^2) &= \frac{(2\pi\mu)^{4-D}}{i\pi^2} \int d^D l \frac{1}{l^2 - m^2} \\
 &= m^2 \left(\frac{1}{\varepsilon} + 1 - \gamma_E + \log 4\pi - \log \frac{m^2}{\mu^2} + O(\varepsilon) \right) \\
 \text{Scalar1Pt}(m) &= \int d^D l \frac{1}{l^2 - m^2} \\
 &= i\pi^2 m^2 \left(\frac{1}{\varepsilon} + 1 - \gamma_E - \log \pi - \log m^2 + O(\varepsilon) \right) \\
 &= \frac{i\pi^2}{(2\pi\mu)^{4-D}} A_0(m^2)
 \end{aligned}$$

Two-Point Functions

$$\begin{aligned}
 B_0(q^2; m_1^2, m_2^2) &= \frac{(2\pi\mu)^{4-D}}{i\pi^2} \int d^D l \frac{1}{(l^2 - m_1^2)((l+q)^2 - m_2^2)} \\
 &= \frac{1}{\varepsilon} + 2 - \gamma_E + \log 4\pi - \log \frac{m_1 m_2}{\mu^2} + \frac{m_1^2 - m_2^2}{q^2} \log \frac{m_2}{m_1} \\
 &\quad + \frac{1}{2} \left(1 + \frac{m_1^2 - m_2^2}{q^2} \right) x_1 (\log(1 - x_1) - \log(1 + x_1) + i\pi) \\
 &\quad + \frac{1}{2} \left(1 - \frac{m_1^2 - m_2^2}{q^2} \right) x_2 (\log(1 - x_2) - \log(1 + x_2) + i\pi) \\
 &= B_0(q^2; m_2^2, m_1^2)
 \end{aligned}$$

$$\text{with } x_1 = \sqrt{1 - \frac{4m_1^2 q^2}{(m_1^2 - m_2^2 + q^2)^2}}, \quad x_2 = \sqrt{1 - \frac{4m_2^2 q^2}{(m_2^2 - m_1^2 + q^2)^2}}.$$

$$\begin{aligned}
 B^\mu(q^2; m_1^2, m_2^2) &= \frac{(2\pi\mu)^{4-D}}{i\pi^2} \int d^D l \frac{l^\mu}{(l^2 - m_1^2)((k+q)^2 - m_2^2)} \\
 &= q^\mu B_1(q^2; m_1^2, m_2^2) \\
 B_1(q^2; m_1^2, m_2^2) &= \frac{1}{q^2} q_\mu B^\mu(q^2; m_1^2, m_2^2) \\
 &= \frac{1}{2q^2} (A_0(m_1^2) - A_0(m_2^2) - (q^2 + m_1^2 - m_2^2) B_0(q^2; m_1^2, m_2^2)) \\
 &= -B_0(q^2; m_1^2, m_2^2) - B_1(q^2; m_2^2, m_1^2)
 \end{aligned}$$

$$\begin{aligned}
 \text{Scalar2Pt}(q, m_1, m_2) &= \int d^D l \frac{1}{((l+q)^2 - m_1^2)(l^2 - m_2^2)} \\
 &= i\pi^2 \left(\frac{1}{\varepsilon} + 2 - \gamma_E - \log \pi + \dots \right) \\
 &= \frac{i\pi^2}{(2\pi\mu)^{4-D}} B_0(q^2; m_1^2, m_2^2)
 \end{aligned}$$

$$\text{OneLoop2Pt}(p_0, p_\perp, q, m_1, m_2) = \int d^D k \frac{l_0^{p_0} l_\perp^{p_\perp}}{((l+q)^2 - m_1^2)(l^2 - m_2^2)}$$

$$\begin{aligned}
 \text{OneLoopTens2Pt}(1, q, m_1, m_2) &= \int d^D l \frac{l^\mu}{((l+q)^2 - m_1^2)(l^2 - m_2^2)} \\
 &= \frac{q^\mu}{\sqrt{q^2}} \text{OneLoop2Pt}(1, 0, q, m_1, m_2) \\
 &= \frac{q^\mu}{2q^2} (\text{Scalar1Pt}(m_2) - \text{Scalar1Pt}(m_1) \\
 &\quad - (q^2 - m_1^2 + m_2^2) \text{Scalar2Pt}(q, m_1, m_2)) \\
 &= \frac{i\pi^2}{(2\pi\mu)^{4-D}} q^\mu B_1(q^2; m_2^2, m_1^2)
 \end{aligned}$$

Three-Point Functions

$$\begin{aligned}
 C_0(q_1^2, q_2^2, q_3^2; m_1^2, m_2^2, m_3^2) \\
 &= \frac{(2\pi\mu)^{4-D}}{i\pi^2} \int d^D l \frac{1}{(l^2 - m_1^2)((l+q_1)^2 - m_2^2)((l+q_1+q_2)^2 - m_3^2)}
 \end{aligned}$$

$$\begin{aligned}
 \text{Scalar3Pt}(q_{10}, q_{20}, q_{21}, m_1, m_2, m_3) &= \int d^D l \frac{1}{((l+q_1)^2 - m_1^2)((l+q_2)^2 - m_2^2)(l^2 - m_3^2)} \\
 &= \frac{i\pi^2}{(2\pi\mu)^{4-D}} C_0(q_1^2, (q_2 - q_1)^2, q_2^2; m_3^2, m_1^2, m_2^2)
 \end{aligned}$$

Glossary

double dispatch: Run-time polymorphism, i. e. the ability to choose one of possibly several functions to call based on the dynamic type of some or all function arguments, is a feature provided to some degree by all object-oriented programming languages.

C++ only offers *single* dispatch as a part of the language. The virtual member function mechanism selects a method at run time according to the dynamic type of *one* (implicit) function argument: the receiving object (as opposed to ordinary function overloading where the selection takes place at compile time based on the static types of the arguments).

It is often useful, however, to have a virtual function chosen by the dynamic types of two (double dispatch) or, in general, all of its arguments (multiple dispatch), especially in cases where it is not obvious whether an interaction between two objects *A* and *B* should be implemented as a method of *A* or a method of *B*.

Some languages such as the Common Lisp Object System provide built-in support for multiple dispatch. In C++ this has to be recreated manually (see [Alex 2001, chapter 11] for some example implementations).

first-class: In a programming language, a data type is said to be *first-class* when objects of the type can be created and manipulated at run time. This also usually implies that values of this type can be stored in variables, passed to and from functions, etc. but the exact set of operations available for first-class objects depend on the language being used.

For example, in C++ functions are not first-class objects but pointers to functions are. The only operations possible on functions in C++ are invocation, and taking their address. Other programming languages such as SMALLTALK and LISP allow the use of functions (or closures, i. e. functions together with their set of variable bindings) as first-class data.

flyweight: A shared object used simultaneously in multiple contexts but acting as an independent object in each one.

function object/functor: In C++, an object that can be called as a function. This includes proper C++ functions, function pointers, constructors, and instances of classes overloading `operator()` (the function-call operator). Sometimes, the use of the term is limited to the latter kind.

garbage collection: The process of automatically reclaiming the memory space taken up by objects which are no longer referenced. There exist many different methods of determining for which objects this is the case, and when to reclaim the memory used

by these objects. GiNaC utilizes a reference counting garbage collector for algebraic objects in which each object contains a counter of the number of references to it. Once this number reaches zero, the object is deleted.

hash table: A hash table is an implementation of an associative array (a key→value mapping) with on average constant-time complexity for element insertions, deletions, and look-ups. The (key,value) pairs are stored in a linear table, indexed by a hash value obtained by applying a *hash function* $h : K \rightarrow \mathbb{N}$ to the keys $k \in K$. Each table entry is called a *bucket*.

The hash function can be arbitrary, but should be chosen in a way as to reduce the frequency of *hash collisions* ($h(k_1) = h(k_2)$ for $k_1 \neq k_2$) as far as possible. Also, if the calculation of the hash function is expensive compared to comparisons of the key values, a linear or binary search in a (key,value) table may offer better performance than a hash table for small numbers of elements.

There is extensive literature dealing with hash tables. See, for example [Knut 1973, chapter 6].

mixin class: A class intended to be composed with other classes through inheritance.

policy class: By encapsulating a family of related algorithms sharing a common interface (the *policy*) in separate classes, one for each algorithm, the behavior of a client object or class template can be customized simply by selecting the appropriate policy class. This corresponds to the *Strategy* pattern of [GHJV 1995].

smart pointer: In a language such as C++ that requires manual management of the storage space occupied by objects it is often desirable to lower the chance of memory leaks and other programming errors by creating classes that behave syntactically and semantically like built-in pointers but implement some kind of automated resource management behind the scenes. The STL `auto_ptr<>` template is an example for such a “smart” pointer.

Smart pointers can implement a variety of storage management policies such as *deep copy* (copying the pointer also copies the pointed-to object), *destructive copy* (copying the pointer relinquishes the ownership of the pointed-to object from the original pointer; used by `auto_ptr`), and *reference counting*.

Bibliography

- [Alex 2001] Andrei Alexandrescu: *Modern C++ Design: Generic Programming and Design Patterns Applied*; Addison-Wesley, 2001
- [BaDo 2002] Christian Bauer, Do Hoang Son: *One-loop integrals with XLOOPS-GiNaC*; Comp. Phys. Commun. 114 (2002) 154; arXiv: hep-ph/0102231
- [Baue 2000] Christian Bauer: *Der xloops-Algorithmus zur Berechnung von Feynman-Graphen in C++*; diploma thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dip-cbauer.ps.gz>
- [BaZa 2004] Roberto Bagnara, Alessandro Zaccagnini: *Checking and Bounding the Solutions of Some Recurrence Relations*; technical report, Quaderno 344 (2004), Department of Mathematics, University of Parma; see URL: <http://www.cs.unipr.it/Publications/Abstracts/Q344>
- [BDIPS 1994] Edward E. Boos, Mikhail N. Dubinin, Viacheslav A. Ilyin, Alexander E. Pukhov, Victor I. Savrin: *CompHEP – Specialized Package for Automatic Calculations of Elementary Particle Decays and Collisions*; SNUTP 94-116; INP MSU-94-36/358; arXiv: hep-ph/9503280
- [BDJ 2001] Manfred Böhm, Ansgar Denner, Hans Joos: *Gauge Theories of the Strong and Electroweak Interaction*; 3rd edition, Teubner, 2001
- [Be 1997] The Be Development Team: *Be Developer’s Guide*; O’Reilly & Associates, 1997
- [BeSh 2004] A. Bednyakov, A. Sheplyakov: *Two-loop $\mathcal{O}(\alpha_s y^2)$ and $\mathcal{O}(y^4)$ MSSM corrections to the pole mass of the b-quark*; arXiv: hep-ph/0410128
- [BFFK 1997] Lars Brücher, Johannes Franzkowski, Alexander Frink, Dirk Kreimer: *Introduction to XLOOPS*; Nucl. Instrum. Meth. A389 (1997) 323–342; arXiv: hep-ph/9611378
- [BFK 1994] Lars Brücher, Johannes Franzkowski, Dirk Kreimer: *Loop integrals, R functions and their analytic continuation*; Mod. Phys. Lett. A9 (1994) 2335–2346; arXiv: hep-th/9307055
- [BFK 1995] Lars Brücher, Johannes Franzkowski, Dirk Kreimer: *A New Method for Computing One-Loop Integrals*; Comp. Phys. Commun. 85 (1995) 153–165; arXiv: hep-ph/9401252

- [BFK 1997] Lars Brücher, Johannes Franzkowski, Dirk Kreimer: *oneloop 2.0 – A Program Package Calculating One-Loop Integrals*; Comp.Phys. Commun. 107 (1997) 281–292; arXiv: hep-ph/9709209
- [BFK 1998] Lars Brücher, Johannes Franzkowski, Dirk Kreimer: *XLoops: Automated Feynman Diagram Calculation*; Comp.Phys. Commun. 115 (1998) 140–160
- [BFK 2002] Christian Bauer, Alexander Frink, Richard Kreckel: *Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language*; J.Symbolic Computation 33 (2002) 1–12; arXiv: cs-sc/0004015
- [BFK 2003] Christian Bauer, Alexander Frink, Richard Kreckel: *GiNaC*; in: J. Grabmeier, E. Kaltofen, V. Weispfenning (ed.): *Computer Algebra Handbook*; Springer, 2003
- [BFKV 2004] Christian Bauer, Alexander Frink, Richard Kreckel, Jens Vollinga: *GiNaC – An open framework for symbolic computation within the C++ programming language*; see URL: <http://www.ginac.de/tutorial/>
- [BKK 2002] Isabella Bierenbaum, Richard Kreckel, Dirk Kreimer: *On the Invariance of Residues of Feynman Graphs*; J.Math.Phys. 43 (2002) 4721–4740; arXiv:hep-th/0111192
- [BrRa 1997] Rene Brun, Fons Rademakers: *ROOT – An Object Oriented Data Analysis Framework*; Proceedings AIHENP'96 Workshop, Lausanne, September 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81–86; see URL: <http://root.cern.ch>
- [Brüc 1993] Lars Brücher: *Algorithmen zur Berechnung von Einschleifen-Integralen im Standardmodell*; diploma thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dip-bruecher.ps.gz>
- [Brüc 1997] Lars Brücher: *Automatische Berechnung von Strahlungskorrekturen in perturbativen Quantenfeldtheorien*; Ph.D. thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dis-bruecher.ps.gz>
- [Capr 1999] H. Caprasse; *CANTENS – A Package for Manipulations and Simplifications of Indexed Objects*; see URL: <http://www.zib.de/Symbolik/reduce/moredocs/cantens.pdf>
- [CCS 1999] Arjeh M. Cohen, Hans Cuypers, Hans Sterk (eds.): *Some Tapas of Computer Algebra*, Algorithms and Computation in Mathematics Vol. 4, Springer, 1999
- [CKK 1995] Andrzej Czarnecki, Ulrich Kilian, Dirk Kreimer: *New representation of two-loop propagator and vertex functions*; Nucl.Phys. B433 (1995) 259–275; arXiv: hep-ph/9405423

-
- [Chis 1963] J. S. R. Chisholm, *Il Nuovo Cimento* X 30 (1963) 426
- [ChTk 1981] Konstantin G. Chetyrkin, Fyodor V. Tkachov: *Integration by Parts: The Algorithm to Calculate β -Functions in 4 Loops*; Nucl. Phys. B192 (1981) 159–204
- [Coll 1984] John C. Collins: *Renormalization*; Cambridge University Press, 1984
- [DaT 1992] Andrey I. Davydychev, Jan B. Tausk: *Two-Loop Self-Energy Diagrams With Different Masses and the Momentum Expansion*; Nucl. Phys. B397 (1992) 123–142
- [Denn 1993] Ansgar Denner: *Techniques for the Calculation of Electroweak Radiative Corrections at the One-Loop Level and Results for W-physics at LEP200*; Fortschr. Phys. 41 (1993) 4, 307–420
- [DeSa 1990] Ansgar Denner, Thomas Sack: *Renormalization of the Quark Mixing Matrix*; Nucl. Phys. B347 (1990) 203–216
- [DGBEG 1996] Elise de Doncker, Ajay Gupta, Jay Ball, Patricia Ealy, Alan Genz: *ParInt: A Software Package for Parallel Integration*; Proc. of the 10th ACM International Conference on Supercomputing, 149–156
- [Ditt 1998] Stefan Dittmaier; *Weyl-van-der-Waerden formalism for helicity amplitudes of massive particles*; CERN-TH/98-143; arXiv: hep-ph/9805445
- [Do 2003] Do Hoang Son; *Feynman loop integrals and their automatic computer-aided evaluation*; Ph.D. thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dis-dhson.pdf>
- [Fran 1992] Johannes Franzkowski: *Analytische Berechnung von Einschleifen-Integralen in perturbativer Quantenfeldtheorie*; diploma thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dip-franzkowski.ps.gz>
- [Fran 1997] Johannes Franzkowski: *Virtuelle Strahlungskorrekturen im Standardmodell der Elementarteilchenphysik*; Ph.D. thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dis-franzkowski.ps.gz>
- [Frin 1996] Alexander Frink: *Massive Zwei-Loop Vertexfunktionen*; diploma thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dip-frink.ps.gz>
- [Frin 2000] Alexander Frink: *Computer-algebraische und analytische Methoden zur Berechnung von Vertexfunktionen im Standardmodell*; Ph.D. thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dis-frink.ps.gz>
- [GCL 1992] Keith O. Geddes, Stephen R. Czapor, George Labahn: *Algorithms for Computer Algebra*; Kluwer, Norwell, Massachusetts

- [Gett 1989] James Gettys, Robert W. Scheifler, Ron Newman: *Xlib – C Language X Interface*; MIT X Consortium Standard, X Version 11, Release 4; MIT, Massachusetts, 1989
- [GHJV 1995] Erich Gamma, Richard Helms, Ralph Johnson, John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*; Addison-Wesley, Reading, 1995
- [Gosp 1978] R. W. Gosper: *Decision Procedure for Indefinite Hypergeometric Summation*; Proc. Nat. Acad. Sci. USA 75 (1978) 40–42
- [Goto 2002] Masaharu Goto: *The CINT C/C++ Interpreter* (version 5.15); see URL: <http://root.cern.ch/root/Cint.html>
- [Grun 1996] Dominik Gruntz: *On Computing Limits in a Symbolic Manipulation System*; Ph. D. thesis, Swiss Federal Institute of Technology, Zürich
- [Guzm 2002] Joel de Guzman: *The Spirit Parser Framework*; see URL: <http://spirit.sourceforge.net/>
- [HaKr 2000] Bruno Haible, Richard Kreckel: *CLN, a Class Library for Numbers* (Version 1.1); see URL: <http://www.ginac.de/CLN/>
- [Hoei 2002] Mark van Hoeij: *Factoring polynomials and the knapsack problem*; Journal of Number Theory 95 (2002) 167–189
- [IKKKST 1993] Tadashi Ishikawa, Toshiaki Kaneko, Kiyoshi Kato, Setsuya Kawabata, Yoshimisu Shimizu, T. Tanaka: *GRACE Manual, Version 1.0*; KEK Report 92-19; Comp. Phys. Commun. 92 (1993) 127–152
- [ISO 1998] ISO/IEC 14882:1998(E): *Programming Languages – C++*; American National Standards Institute, 1998
- [Jame 1987] Geoffrey James: *The Tao of Programming*; InfoBooks, Santa Monica, 1987
- [Jege 2001] Fred Jegerlehner: *Facts of life with γ_5* ; Eur. Phys. J. C18 (2001) 673; arXiv: hep-th/0005255
- [KKS 1992] Jürgen D. Körner, Dirk Kreimer, Karl Schilcher: *A Practicable γ_5 Scheme in Dimensional Regularization*; Z. Phys. C54 (1992) 503–512
- [Knut 1973] Donald E. Knuth: *The Art of Computer Programming; Vol. 3: Sorting and Searching*; Addison-Wesley, Reading, 1973
- [Krec 1998] Richard Kreckel: *Parallelization of Adaptive MC Integrators*; Comp. Phys. Commun. 106 (1998) 258–266; arXiv: physics/9710028
- [Krec 1997] Richard Kreckel: *Irreduzible Zwei-Loop-Beiträge zu den Prozessen $\gamma\gamma \rightarrow \pi\pi$ und $\eta \rightarrow \pi\gamma\gamma$* ; diploma thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dip-kreckel.ps.gz>

-
- [Krec 2002] Richard Kreckel: *Algorithmische Methoden zur Berechnung von Vierbeinfunktionen*; Ph.D. thesis, Mainz; see URL: <http://www.thep.physik.uni-mainz.de/Publications/theses/dis-kreckel.ps.gz>
- [Krei 1990] Dirk Kreimer: *The γ_5 Problem and Anomalies: A Clifford Algebra Approach*; Phys. Lett. B237 (1990) 59
- [Krei 1991] Dirk Kreimer: *The master two-loop two-point function – The general case*; Phys. Lett. B273 (1991) 277–281
- [Krei 1992a] Dirk Kreimer: *The two-loop three-point functions: General massive cases*; Phys. Lett. B292 (1992) 341–347
- [Krei 1992b] Dirk Kreimer: *One-loop integrals revisited I: The two-point functions*; Z. Phys. C54 (1992) 667–672
- [Krei 1993a] Dirk Kreimer: *2-loop Integrals in the Standard Model*; Phys. Atom. Nucl. 56 (1993) 1546–1552; arXiv: hep-ph/9212254
- [Krei 1993b] Dirk Kreimer: *One-loop integrals revisited II: The three-point functions*; Int. J. Mod. Phys. A8 (1993) 1797–1814
- [Krei 1993c] Dirk Kreimer: *The Rôle of γ_5 in Dimensional Regularization*; arXiv: hep-ph/9401354
- [Krei 1994] Dirk Kreimer: *Tensor Integrals for Two Loop Standard Model Calculations*; Mod. Phys. Lett. A9 (1994) 1105–1120; arXiv: hep-ph/9312223
- [LaKr 2003] Angelika Langer, Klaus Krefl: *C++ Expression Templates*; C/C++ Users Journal, March 2003, pp. 27–34
- [Lepa 1978] G. Peter Lepage: *A New Algorithm for Adaptive Multidimensional Integration*; J. Comput. Phys. 27 (1978) 192–203
- [LMB 1992] John R. Levine, Tony Mason, Doug Brown: *lex & yacc*; O’Reilly & Associates, 1992
- [Mart 1998] Robert C. Martin: *Acyclic Visitor*; in R. C. Martin, D. Riehle, F. Buschmann (eds.): *Pattern Languages of Program Design, Vol. 3*; Addison Wesley, 1998; see URL: <http://www.objectmentor.com/publications/acv.pdf>
- [MBD 1991] Rolf Mertig, Manfred Böhm, Ansgar Denner, Comp. Phys. Commun. 64 (1991) 345
- [MUW 2002] Sven Moch, Peter Uwer, Stefan Weinzierl: *Nested Sums, Expansion of Transcendental Functions and Multi-Scale Multi-Loop Integrals*; J. Math. Phys. 43 (2002) 3363; arXiv: hep-ph/0110083
- [PaVe 1979] Giampiero Passarino, Martinus J. G. Veltman; Nucl. Phys. B160 (1979) 151

- [Pert 2001] Pertti Lounesto: *Clifford Algebras and Spinors*; 2nd edition, London Mathematical Society Lecture Note Series 286, Cambridge University Press
- [PeSc 1995] Michael E. Peskin, Daniel V. Schroeder: *Introduction to Quantum Field Theory*; Addison-Wesley, Reading, 1995
- [Post 1997] Peter Post: *Elektromagnetische und schwache Vektorformfaktoren des pseudoskalaren Meson-Oktetts zur Ordnung p^6 der chiralen Störungstheorie*; Ph.D. thesis, Mainz; see URL: <http://wwwthep.physik.uni-mainz.de/Publications/theses/dis-peterpost.ps.gz>
- [PWZ 1997] Marko Petkovsek, Herbert S. Wilf, Doron Zeilberger: *A = B*; AK Peters, Wellesley, Massachusetts
- [Rati 1997] RATIONAL Software Corporation: *UML Notation Guide* (Version 1.1); see URL: <http://www.rational.com/>
- [Shou 2002] Victor Shoup: *NTL – A Library for Doing Number Theory* (Version 5.3.2); see URL: <http://shoup.net/ntl/>
- [ShSt 1998] Tan Kiat Shi, Willi-Hans Steeb: *SymbolicC++ – An Introduction to Computer Algebra using Object-Oriented Programming*; Springer, 1998
- [Stoy 2004] Herbert Stoyan: *LISP History*; see URL: <http://www8.informatik.uni-erlangen.de/html/lisp-enter.html>
- [tHVe 1979] Gerardus 't Hooft, Martinus J. G. Veltman: *Scalar One-Loop Integrals*; Nucl. Phys. B153 (1979) 365–401
- [Veld 1995] Todd Veldhuizen: *Expression Templates*; C++ Report, Vol. 7 No. 5 June 1995, pp. 26–31
- [Veld 2002] Todd Veldhuizen et al.: *Blitz++ library* (Version 0.6); see URL: <http://www.oonumerics.org/blitz/>
- [Wein 2002] Stefan Weinzierl: *Computer Algebra in Particle Physics*; arXiv: hep-ph/0209234
- [Wein 2004a] Stefan Weinzierl: *Expansion around half-integer values, binomial sums and inverse binomial sums*; arXiv: hep-ph/0402131
- [Wein 2004b] Stefan Weinzierl: *gTybalt – a free computer algebra system*; Comp. Phys. Commun. 156 (2004) 180–198; arXiv: cs-sc/0304043
- [Zass 1969] Hans Zassenhaus: *On Hensel Factorization*; Journal of Number Theory 1 (1969) 291–311