MZ-TH/02-01
Talk given at Computer Particle Physics (CPP 2001), 27-30 November 2001, Tokyo.

using namespace GiNaC;

Richard B. Kreckel <Richard.Kreckel@Uni-Mainz.DE>
WA ThEP, Inst. f. Physik, Johannes Gutenberg University, Mainz, Germany

February 10, 2002

Abstract

Some of the experiences gained while designing, implementing and actually using the GiNaC system for symbolic computation in C++ are outlined. Difficulties and pitfalls for users migrating from Form are contrasted and some of the benefits and liabilities are sketched shortly.

This workshop provided us with some heated discussion on the pros and cons of various competing programming languages and programming paradigms. Thus, we were led to reflect all the three major sources for the infamous "language wars": First, the nuisance of once popular but now archaic languages that still need to be supported due to existing legacy code. Second, barriers of incompatibilities that hinder migration and interfacing even from one "modern" language to another. And last, but not least, ignorance about the competing language's advantages and benefits.

In the light of this, let us briefly review why C++ has been chosen as the language of implementation for the GiNaC [1] symbolic system. The old xloops project [2] more or less failed to be able to support its own codebase. Besides several architectural shortcomings one of the main culprits is the Maple language and there, the largest contribution to this failure comes from the dynamic scoping. Any modern language with lexical scoping would be a remedy to this but the two major SAC systems are basically stuck with their scoping rules because of their user community's codebase.[1] Introducing symbols is possible in any object-oriented language, one just has to declare a class of them and carefully specify its interactions. When these interactions are entirely clear it is not desirable to have to write addition of symbols as `x.add(y)` (Java), as `add(x,y,&z)` (C), as `(+ x y)` (Lisp) when we can write in plain algebraic syntax `x+y` as in C++ using an overloaded operator. While it was clear that C++ is not without its faults[2] we also have to take a pragmatic attitude. C++ is the *lingua franca* of scientific computation, especially among our experimental colleagues who we must not forget. And the existence of an international standard [3] guards us from subtle changes in the language that render our labor useless from one day to the other. It also makes us independent from compiler vendors – the sad fate of Pascal which was both rescued and eaten by a reputable company comes into mind. It also allows us to blur the boundary between numerical and symbolic computation. The existence of powerful data structures facilitates the design of advanced numeric algorithms [4]. We have all witnessed circumstances when the generality of the symbolic domain needs to be abandoned in favor of low-level bit-manipulation facilities.

This paper will first describe the algebraic behavior of the GiNaC system in particular with respect to where it differs from Form [5], then present the implementation of a physical application before coming to some conclusions.

1 GiNaC's Symbolic Philosophy

GiNaC was designed to be a strong symbolic engine for the needs arising in high energy physics. Memory-wise it does not extend on the operating system's VM as it is done by Form. Instead, it pro-

[1] It is interesting, though, to note that the SAC underdog MuPAD recently underwent the switch from dynamical to lexical scoping notwithstanding the considerable amount of work involved in converting their whole library to the new rules.

[2] It took mankind 200 years to learn how to build good pianos. It seems naive to believe that we already know everything about language design!

vides more algorithmic support. From C++ it inherits the support for large-scale programming as well as portability. It is open and free, as defined by the GNU General Public License (GPL).

When considering computations in the symbolic domain, the need to answer questions about representation becomes pressing. Consider something as simple as $(x+y)^2$, a polynomial in x and y with the coefficients assumed to be in the integral domain of integers. If we compare it to $x^2+2xy+y^2$ we can write this using GiNaC:

```
symbol a("a"), b("b");
ex f1 = pow(a + b, 2);
ex f2("a^2 + 2*a*b + b^2", lst(a, b));
cout << f1 - f2 << endl;
```

The output is $(a+b)^2 - a^2 - 2ab - b^2$. This is because canonicalization of polynomials does not involve expanding them. One has to call `.expand()` explicitly in order to obtain zero. In contrast, Form, Fermat and Singular internally always represent expressions with ring-operations fully expanded. In Form, the code snippet

```
Symbol a,b;
Local f1 = (a + b)^2;
Local f2 = a^2 + 2*a*b + b^2;
Local d = f1 - f2;
print;
.end
```

prints out zero immediately.

Just as in Maple or Mathematica, when two terms on the same level of the expression-tree match syntactically, the anonymous evaluator kicks in to simplify them. Thus, $(a+b)^2 - (a+b)^2$ immediately simplifies to zero. Here are some other examples and and what the anonymous evaluator makes out of them:

$$\begin{array}{rcl} x+x & \longrightarrow & 2x \\ (3x^2+y)/(3x^2+y) & \longrightarrow & 1 \\ (1+2x-3x^2+4x^3)*3 & \longrightarrow & 3+6x-9x^2+12x^3 \\ (3x^2+y)/(6x^2+2y) & \longrightarrow & (3x^2+y)/(6x^2+2y) \end{array}$$

In order to cancel common factors from numerator and denominator in the last expression, one has to explicitly call the method `.normal()`. It is similar to Maple's function `normal()`, Mathematica's `Together[]` or Macsyma's `radsimp()`. One can think of it as the equivalent of `.expand()` acting on elements of division fields instead of rings. The fact that we have to call a function explicitly makes quotients of polynomials behave in a symmetric way to polynomials. It also leaves somewhat more choice to the programmer. To illustrate this, the fraction $(x^{101}-1)/(x-1)$ can also be written as $x^{100}+x^{99}+\cdots+x+1$, which can hardly be considered to be simpler. Also, the cancellation of GCDs in numerator and denominator might benefit from unexpanded polynomials. In the fraction

$$\frac{(x^3-3y^2x^2+2xy-6y^3)(2x-2)}{(x-1)(x-3y^2)} \quad \overset{\texttt{.normal()}}{\longrightarrow} \quad 2x^2+4y$$

an early cancellation of $(x-1)$ can drastically reduce the complexity of computing the GCD. This benefit is not purely theoretical. Robert Lewis and Michael Wester [6] have suggested a set of symbolic benchmarks, one of which (test N, in their paper) involves testing for zero a large multivariate polynomial when evaluated at certain rational functions. Expanding the numerator alone to check if it gives zero would require enormous amounts of storage but one can successively cancel factors and by this, GiNaC comes up with the result in about ten minutes. It seems to be the second system to be able to compute this expression.

Notice, that in the above code-samples, we have used two sorts of constructors for expressions. There was the syntax known from C/C++ (like `pow(a+b,2)`) and there was a constructor from parsable strings (like `"a^2+2*a*b+b^2"`). As a rule, GiNaC's default form of output is compatible with the latter one: printing and reading the resulting expressions again is an operation satisfying idempotency. Since the output is similar to the one know from Maple, Reduce or Form it is even possible to share application-specific algorithms without rewriting source code.

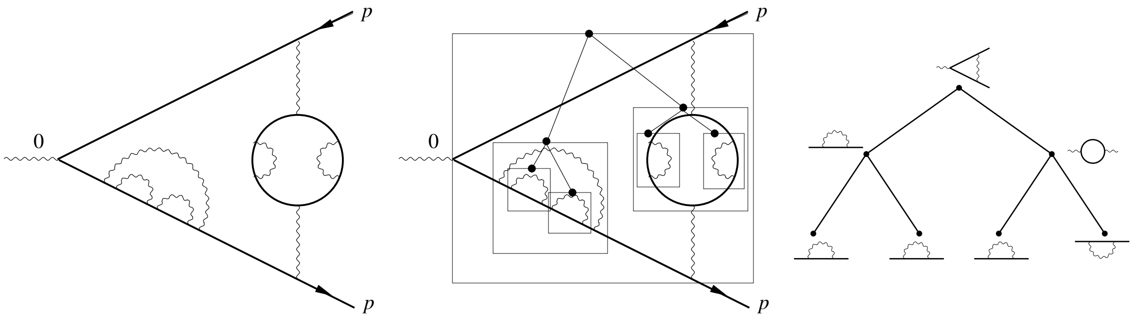
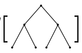


Figure 1: A Feynman graph from massless Yukawa theory, its Zimmermann forest and the emergence of the associated decorated rooted tree.

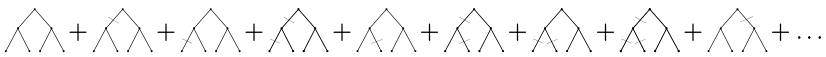
2 A Case Study: Counterterms via Antipode in Hopf Algebra of Rooted Trees

In [7] a program was presented that computes the counterterms in dimensional regularization in a certain class of multi-loop diagrams by using the Hopf algebra of rooted trees. The graphs under consideration comprise all nested single loops in massless Yukawa theory and massless QED. The nested nature currently rules out overlapping as well as crossed topologies. It is not our aim to discuss the Hopf algebra of Feynman graphs here, not even the Hopf algebra of rooted trees. Suffice to say that the rooted trees arise naturally from the nested Feynman graphs as shown in figure 1. Topologists will recognize the rooted tree as the inverse topology of the fully grown Zimmermann forest. We'll outline the actual implementation of these computations here.

What we want to compute are the counterterms in the ε-expansion which happen to be the antipodes in the underlying Hopf algebra. Thus,

$S[$  $]$ $$= a_7\varepsilon^{-7} + a_6\varepsilon^{-6} + a_5\varepsilon^{-5} + a_4\varepsilon^{-4} + a_3\varepsilon^{-3} + a_2\varepsilon^{-2} + a_1\varepsilon^{-1}$$

turns out to be computable via the sum over all possible cuts in the decorated rooted tree, i.e:



where a cut tells us that the part of the tree underneath it is supposed to be truncated to its divergent part and at each of the tree's node a characterizing function is inserted and series expanded. For instance, the one-loop fermion self-energy with insertions at its two internal propagators demands expanding

$$\Sigma_{i,j} \equiv \frac{1}{2}[F_{i\varepsilon,1+j\varepsilon} + F_{1+i\varepsilon,1+j\varepsilon} - F_{1+i\varepsilon,j\varepsilon}] \tag{1}$$

with $F_{a,b}$ given by

$$F_{a,b} \equiv F_{b,a} = \frac{\Gamma(2-a-\varepsilon)\,\Gamma(2-b-\varepsilon)\,\Gamma(a+b-2+\varepsilon)}{\Gamma(a)\Gamma(b)\Gamma(4-a-b-2\varepsilon)}.$$

Not minding the indices, we'll sketch how to naturally write a program that traverses the trees bottom-up and thus computes the antipode in C++ in a hope that the elegance of the solution speaks for itself. We first define an abstract base class for all kinds of possible decorations

```
class decoration {
protected:
    pair<unsigned, unsigned> indices;
public:
    // ctors, dtors, boilerplate...
    virtual const ex evaluate(const symbol &x, const unsigned deg) const = 0;
};
```

where the constructors, destructors and common boilerplate like a `.clone()`-method are left out for brevity. A class `Sigma` then implements the characterizing function (1). Here is its declaration:

```
class Sigma : public decoration {
public:
    // ctors, dtors, boilerplate...
    const ex evaluate(const symbol & x, const unsigned deg) const;
};
```

This is followed by implementations for Vertex graphs (`Gamma`) and vacuum polarization. So far, this allows us to handle the decoration only, not entire trees. For this, we use a node class that incorporates one decoration (`Gamma`, `Sigma`, etc.) and its child nodes. Since only the multiplicity of the child nodes matters, not its order, they form a multiset. We don't have to code the `multiset<T>` structure, everything is there in the Standard Template Library (STL):

```
class node {
    decoration *deco;                        // allocated on the heap
    multiset<pair< node, bool > > children;  // boolean tags cut or uncut
public:
    // ctors, dtors, delegtors, etc...
    const ex evaluate(const symbol &x, const unsigned deg) const;
};
```

The decoration is going to be stored on the heap because we want to delegate the call of `.evaluate()` to the correct characterizing function using dynamic type dispatch ("late binding"). The tree's `.evaluate()` is going to return a series expansion of the characterizing function in some symbol (the dimensional regularization parameter used) to some degree, by scanning all its children bottom-up and truncating the cut branches (with a function called `div_part()` defined elsewhere in a straightforward fashion):

```
const ex node::evaluate(const symbol &x, const unsigned deg) const
{
    ex product = 1;
    for (multiset<pair< node, bool > >::const_iterator i=children.begin();
                                                   i!=children.end(); ++i) {
        if (i->second == false)
            product *= i->first.evaluate(x,deg).series(x==0,deg).expand()
        else
            product *= -div_part(i->first.evaluate(x,deg).series(x==0,deg).expand());
    }
    return (product * deco->evaluate(x,deg));  // Here is the type dispatch
}
```

This results in very flexible and easily maintained code. The only problem is that Reduce and Maple would perform much better because of their internal remembering functions. After all, the branches in our rooted trees are traversed and evaluated multiple times. There is some point in making functions remember, especially if the callee has multiple callers. Frequently, however, it causes problems because the control when to forget things is pushed outside the reach of the caller and putting the results in a database at the callers side is preferable. In our case, we can readily build a toy-database by using the associative container `map<Key, Value>` from the STL:

```
const ex node::evaluate(const symbol &x, const unsigned deg) const
{
    static map<node, ex> catalog;    // lookup-table
    ex product = 1;
    for (multiset<pair< node, bool > >::const_iterator i=children.begin();
                                                   i!=children.end(); ++i) {
        map<node, ex>::iterator pos = catalog.find(i->first);
        if (pos==catalog.end()) {
            pos = catalog.insert(map<node,ex>::value_type(
                     i->first,
                     i->first.evaluate(x,deg).series(x==0,deg).expand())
                  ).first;
        }
        if (i->second == false)
            product *= pos->second;
        else
            product *= -div_part(pos->second,x,grad);
    }
    return (product * deco->evaluate(x,deg));
}
```

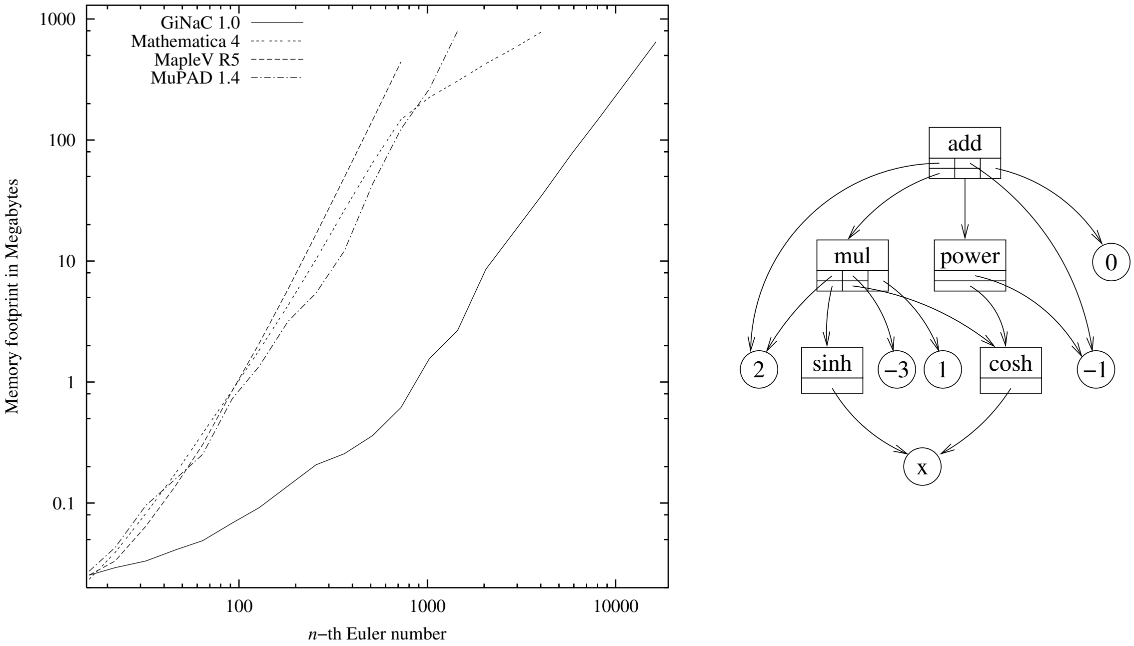


Figure 2: Memory footprint of several SAC systems for the computation of Euler numbers E_n on an x86 architecture (left) and GiNaC's representation tree of $\frac{d^2}{dx^2}(1/\cosh(x)) = 2\sinh(x)^2/\cosh(x)^3 - 1/\cosh(x)$ as it appears when computing E_2 showing a good degree of sharing.

This augmented `.evaluate()` method then shows an excellent performance and it is trivial to free the memory used up by our catalog of return values when this need arises.

The evaluated Laurent series suffer from a huge intermediate expression swell. This is so because the Laurent series arise from products and sums of many Γ-functions with Euler-Mascheroni constants all over the place. In the case of massless Yukawa theory, `.expand()` sorts the terms to a concise form. In massless QED, however, `.expand()` is not enough because of the inclusion of a symbolic gauge parameter ξ which appears in coefficient denominators. Here, we must call `.normal()` instead. For the graph from figure 1 the result turns out to be

$$\begin{aligned}
S[\;] \quad = \quad & \left(\tfrac{1}{4032}\right)\mathrm{Tr}(\mathbb{1})\,\varepsilon^{-7} \\
& +\left(\tfrac{-1}{1008}\right)\mathrm{Tr}(\mathbb{1})\,\varepsilon^{-6} \\
& +\left(\tfrac{23}{26880}\right)\mathrm{Tr}(\mathbb{1})\,\varepsilon^{-5} \\
& +\left(\tfrac{-1}{13440}\zeta(3)+\tfrac{59}{40320}\right)\mathrm{Tr}(\mathbb{1})\,\varepsilon^{-4} \\
& +\left(\tfrac{-1}{8960}\zeta(4)+\tfrac{1}{1792}\zeta(3)+\tfrac{-179}{64512}\right)\mathrm{Tr}(\mathbb{1})\,\varepsilon^{-3} \\
& +\left(\tfrac{-1}{1920}\zeta(5)+\tfrac{3}{3584}\zeta(4)+\tfrac{-193}{53760}\zeta(3)+\tfrac{11}{26880}\right)\mathrm{Tr}(\mathbb{1})\,\varepsilon^{-2} \\
& +\left(\tfrac{-1}{896}\zeta(6)+\tfrac{1}{256}\zeta(5)+\tfrac{-193}{35840}\zeta(4)+\tfrac{191}{10752}\zeta(3)+\tfrac{-1}{4480}\zeta(3)^2+\tfrac{-379}{28672}\right)\mathrm{Tr}(\mathbb{1})\,\varepsilon^{-1}.
\end{aligned}$$

whereas the unexpanded version would fill about 1500 pages when printed. This can be computed in a few seconds. In practice, antipodes of nine-loop graphs in massless QED are still computationally feasible but may require several hours and hundreds of megabytes.

3 Conclusion

GiNaC provides more fundamental algorithmic support than Form does. However, in contrast to Form, its dependence on the VM makes it restricted to the amount of memory provided by the machine. Since the speed-gap between RAM-access and disk-access has widened and seems to continue widening, we have not bothered about file-based computations. Rather, our focus has been to use good old reference-counting in order to deplete multiple copies of algebraic objects from memory. Sometimes, this bears surprising results. An example is given in figure 2. There, we compute the nth Euler numbers by a brute-force application of their definition as Taylor coefficients in

$$\frac{1}{\cosh(x)} \equiv \sum_{n=0}^{\infty} E_n \frac{t^n}{n!}.$$

It may be observed, how the memory footprint for storing the intermediate expressions of $\cosh(x)$ and $\sinh(x)$ pseudofunctions varies by an order of magnitude between various systems. The analysis of the internal representation of E_2 shows how multiply occuring terms in the derivative are successfully "recycled".

The GiNaC system has reached a mature version 1.0 in November 2001. We consider it feature-complete as projected initially and plan to be careful not to break binary compatibility for a while. It was developed under GNU/Linux using the C++ compiler from the GCC but is portable to other good C++ compilers. However, this has so far never led to better timings so we recommend sticking to `g++`.

Experience has shown that the whole system is best used when making full use of the C++ language. Initially, we had planned to also use GiNaC interactively with interactive loop constructs, conditionals and so forth. It was – and still is – tempting to use a C/C++ interpreter for this purpose because it facilitates the transition from the compiled into the interpreted environment and vice versa. An interface to Cint, written by Masaharu Goto and used extensively in the Root [8] framework for data analysis, was written and is still being maintained. Unfortunately, in our opinion, Cint has still too many limitations to be really useful – despite some initial excitement. We have thus switched back to the traditional write-compile-run development cycle which – when using makefiles – is really not so much different from interactive use.

There is no doubt that the bewildering variety of linguistic structures available in C++ makes it difficult for the novice user to choose the right one for some given problem. Over time, in a large project almost all features will eventually be used and the list of pamphleteers quibbling about this is long. Let us just add one reason why people tend to use all the features: *because it is the right thing to do*. For instance, for writing algorithms that traverse a given symbolic expression-tree we have tried various techniques. Such algorithms must use some sort of RTTI (run time type information) to decide what to do with this or that object. Among other techniques we have tried procedural functions that work recursively but this rarely turned out to be fast, elegant or legible. When we finally learned how to properly use the function-call operator (`operator()`, too often used for subscription purposes only) and to think of actual *function objects* traversing the expression tree by reference this immediately simplified the code to a degree where even algorithmic flaws could be detected. Function objects (also called functors) are a fundamental ingredient of C++ and they make implementing powerful external term rewriting rules in GiNaC quite easy.[3] Besides, because of their ability to carry an arbitrary amount of state in a clean way they are more powerful than what can ever be formulated in a procedural language. The same happened with some other features of C++. In the light of this, instead of warning about language aspects that are potentially dangerous, it should be our duty to advise students which feature to use under which preconditions. For this to be successful, we ourselves need to have a solid judgement of different paradigms based on hands-on experience – not a snap judgement based on hearsay. The theoretical crisis with calculations for existing and future colliders is most likely going to be overcome by an evolutionary process of learning and teaching about computational and mathematical tools, not by proclaiming miracle solutions or steering clear of a learning process.

Acknowledgments

I wish to thank our Japanese hosts for hospitality during the CPP2001 workshop. I am also grateful for Jos Vermaseren giving me valuable insights into Form's state and prospects, Toshiaki Kaneko for talks about graph generation, Thorsten Ohl for educated lessons about functional programming and Elise de Doncker for sharing experience with numerical integrators during the workshop.

[3]This topic could unfortunately not be covered here. The interested reader is referred to [9] and our online documentation in general for actual examples of term rewriting rules implemented using functors.

References

[1] Christian Bauer, Alexander Frink, Richard Kreckel: *Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language*; Journal of Symbolic Computation, **33** (2002), 1-12; cs-sc/0004015

[2] Lars Brücher, Johannes Franzkowski, Dirk Kreimer, *Comput. Phys. Commun.* **115**, (1998) 140-160

[3] American National Standards Institute, *ISO/IEC 14882-1998(E) Programming languages — C++* (1998)

[4] Repository for Scientific Computing in Object-Oriented Languages: `http://www.oonumerics.org/`

[5] Jos A. M. Vermaseren: *New Features of Form*; math-ph/0010025

[6] Robert H. Lewis, Michael Wester: *Comparison of Polynomial-Oriented Computer Algebra Systems*; SIGSAM Bulletin **33/4** (1999), 5-13; see also: `http://www.fordham.edu/~lewis/cacomp.html`

[7] Isabella Bierenbaum, Richard Kreckel, Dirk Kreimer: *On the invariance of residues of Feynman graphs*; hep-th/0111192

[8] Rene Brun, Fons Rademakers, *ROOT - An Object Oriented Data Analysis Framework Proceedings of AIHENP 97*, Laussanne, (1996)

[9] *GiNaC's FAQ*; `http://www.ginac.de/FAQ.html`