J. Symbolic Computation (2002) **33**, 1–12
doi:10.1006/jsco.2001.0494
Available online at http://www.idealibrary.com on IDEAL®

Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language

CHRISTIAN BAUER, ALEXANDER FRINK AND RICHARD KRECKEL

Institute of Physics, Johannes-Gutenberg-University, Mainz, Germany

The traditional split into a low level language and a high level language in the design of computer algebra systems may become obsolete with the advent of more versatile computer languages. We describe GiNaC, a special-purpose system that deliberately denies the need for such a distinction. It is entirely written in C++ and the user can interact with it directly in that language. It was designed to provide efficient handling of multivariate polynomials, algebras and special functions that are needed for loop calculations in theoretical quantum field theory. It also bears some potential to become a more general purpose symbolic package.

1. Introduction

Historically, in the design and implementation of symbolic computation engines, the focus has always been rather on algebraic capabilities than on language design. The need for efficient computation in all fields of science has led to the development of powerful algorithms. Thus, the border line between inexact numerical and exact analytical computation has moved, such that more computation may be done exactly before resorting to numerical methods. This development has had great influence on the working practice in such fields as engineering (e.g. robotics), computer science (e.g. networks), physics (e.g. high energy physics) and even mathematics (from group theory to automatic theorem proving in geometry).

This border line between analytical and numerical methods, however, has quite often remained a painful obstacle in the successful application of computer algebra systems (CASs). Usually, some results are obtained with the help of a CAS and these results are integrated later into some other program. This is not only restricted to numerical results, where one frequently translates the output of some CAS to C or even FORTRAN. It is questionable whether the distinction between one language for writing down symbolical algebra, one for obtaining numerical results and maybe a third one for integrating everything in a graphical user interface has any reason other than an historical one. In our experience it frequently leads to confusion; the `xloops` project (Brücher *et al.*, 1998) has somewhat suffered from this.

The chapter “A Critique of the Mathematical Abilities of CA Systems” in Wester (1999) has a section called “Mathematics versus Computer Science” where some misbehaviours of common CASs are shown. There, the first test tries to find out if a global variable is accessed in some local context, in particular within sums, products, limits and integrals. Of the seven systems tested, only Derive passed the test. Even explicitly declaring a variable to be local does not always spare the programmer surprises: MapleV Releases 3 to 5 for instance do not honour local variables if they are created by concatenating strings using the operator dot (`.`), a feature people often feel tempted to use for elegant subscripting. So created variables may be used as lvalues in assigning a *local* variable within a procedure but the result is a modified *global* variable.† Such violations of scope have repeatedly led to subtle bugs within `xloops`. They are notoriously difficult to disentangle since they go undetected until some other part of the programme breaks.

The general picture is that most currently used CASs are linguistically rather impoverished and put up high obstacles to the design of combined symbolical/numerical/graphical programmes. An incomplete look into the toolchest of a C++ developer throws some light on the features that any professional programmer will miss from common CA systems:

- structured data types like `struct`s and `class`es instead of unnamed lists of lists,
- the object oriented (OO) programming paradigm in general,
- `template`s, which allow for generic (i.e. type-independent) programming even in a strongly typed language,
- the Standard Template Library (STL), which provides convenient classes for many kinds of containers with asymptotically ideal access methods and to a large degree container-independent algorithms (`sort`, etc.) to be instantiated by the programmer,
- modularization facilities like `namespace`s,
- powerful development tools like editors (e.g. with automatic indentation and syntax highlighting), debuggers, visualization tools, documentation generators, etc.,
- flexible error handling with exceptions,
- last, but not least, an established standard (ISO/IEC 14882-1998) which guards the programmer from arbitrary changes made by the manufacturer which break existing code.

Solutions for those problems so far are restricted to allowing calls to CAS functionality from other languages and the already mentioned code generators. At most, bridges are built to cross the gap, but no unification of the two worlds is achieved.

1.1. THE GOAL

Loop calculations in quantum field theory (QFT) are one example of such a combined symbolical and numerical effort. The n-fold nested integrals arising there are solved with specialized methods that demand efficient handling of order 10^3–10^6 symbolic terms. At the one-loop level, Feynman graphs can be expressed completely analytically and so in the early 1990s our group started to build up the program package `xloops` based on Maple. The continuation of `xloops` with Maple up to the two-loop level turned out to be very difficult to accomplish. There were numerous technical issues of coding such as the ones outlined above as well as a nasty restriction built into MapleV of no more than 2^{16} terms in sums.

An analysis of `xloops` showed, however, that only a small part of Maple's capability is actually needed: composition of expressions consisting of symbols and elementary functions, replacement of symbols by expressions, non-commuting objects, arbitrarily sized

†We have been told that Maple6 still suffers from this problem.

integers and rationals and arbitrary precision floats, collecting expressions in equal terms, power series expansion, simplification of rational expressions and solutions of symbolic linear equation systems.

It is possible to express all this directly in C++ if one introduces some special classes of symbols, sums, products, etc. More generally, one wishes to freely pass general expressions to functions and back. Here is an example of how some of these things are actually expressed in C++ using the GiNaC[†] framework:

```
1  #include <ginac/ginac.h>
2  using namespace GiNaC;
3
4  ex HermitePoly(const symbol & x, int n)
5  {
6      const ex HGen = exp(-pow(x,2));
7      // uses the identity H_n(x) == (-1)^n exp(x^2) (d/dx)^n exp(-x^2)
8      return normal(pow(-1,n) * HGen.diff(x, n) / HGen);
9  }
10
11 int main(int argc, char **argv)
12 {
13     int degree = atoi(argv[1]);
14     numeric value = numeric(argv[2]);
15     symbol z("z");
16     ex H = HermitePoly(z,degree);
17     cout << "H_" << degree << "(z) == "
18          << H << endl;
19     cout << "H_" << degree << "(" << value << ") == "
20          << H.subs(z==value) << endl;
21     return 0;
22 }
```

When this program is compiled and called with 11 and 0.8 as command line arguments it will readily print out the 11th Hermite polynomial together with that polynomial evaluated numerically at $z = 0.8$:

```
H_11(z) == -665280*z+2217600*z^3-1774080*z^5+506880*z^7-56320*z^9+2048*z^11
H_11(0.8) == 120773.8855954841959
```

Alternatively, it may also be called with an exact rational second argument 4/5:

```
H_11(z) == -665280*z+2217600*z^3-1774080*z^5+506880*z^7-56320*z^9+2048*z^11
H_11(4/5) == 5897162382592/48828125
```

It calls the subroutine `HermitePoly` with the symbolic variable z and the desired order as arguments. There, the Hermite polynomial is computed in a straightforward way using a Rodrigues representation. The `normal()` call therein cancels the generators `HGen` in numerator and denominator. Note that the operators `*` and `/` have been overloaded to allow expressive construction of composite expressions and that object-style method invocations (*obj*.`f`(*arg*)) as well as function-style calls (`f`(*obj*, *arg*)) are possible. Technically, the whole GiNaC library is hidden in a namespace that needs to be imported (for instance with the `using` directive in line 2) in order to allow easy integration with other packages without potential name clashes. This is just a crude example that invites

[†]GiNaC is a recursive acronym for *GiNaC is Not a CAS*.

obvious refinement like parameter checking or rearranging the polynomial in order to make it less sensitive to numerical rounding errors.

Since pattern matching is something that does not blend very naturally into the context of a declarative language like C++, GiNaC takes care to use term rewriting systems which bring expressions into equivalent canonical forms as far as feasible in an economic way. In addition, specialized transformations may be invoked by the user, for instance an `.expand()` method for fully expanding polynomials or a `.normal()` method for polynomial GCD cancellation. As for instance in Form (Vermaseren, 1991) the user alone is responsible for deciding the order of steps to take in some application, there are only very few rules built into GiNaC. The only kind of pattern matching we want to allow is an atomic one, where inside an expression a symbol (or a list of symbols) is replaced by other expressions in this fashion: `(5*a).subs(a==b)` ⇒ `5*b`. We believe that such a conservative restriction should be acceptable to programmers of large systems since the potential ambiguities introduced by pattern matching and overlapping rules can be rather subtle.

2. The Implementation

The implementation of GiNaC follows an OO philosophy: all algebraic classes that may be manipulated by the system are derived from an abstract base class called `basic`.[†] Some of the classes are atomic (symbols, numbers...), others are container classes (sums, products...). Since at run-time container classes must be flexible enough to store different objects whose size must, however, have already been defined at compile-time, we define the class of all expressions, simply called `ex`. It is a wrapper class that stands outside the class hierarchy and it mainly contains a pointer to some object of the class hierarchy. The container classes thus may be restricted to hold objects of the wrapper class `ex` (Figure 1). Because of this "handle" character, objects of class `ex` are also the ones the user creates most of the time. Most operators have been overloaded to work within this class and they are the most common arguments to the functions in GiNaC. Two obvious drawbacks of this flexibility are the lack of type-safety at compile-time and possible performance losses by additional function calls in method invocations. To some extent, this can be remedied by carefully overloading specialized functions and operators. On the other hand, the interplay between `ex` and `basic` (and all classes derived from it) enables us to implement an efficient memory management using reference counting and copy-on-write semantics: multiply occurring expressions (or subexpressions within an expression tree) are shared in memory and copied only when they need to be modified in one part of the program. This happens in a completely transparent way for the user. In order to create one's own classes managed this way it suffices to derive them from class `basic`.

Table 1 gives an overview of what classes are currently provided by GiNaC. We are now going to describe some of them.

2.1. NUMBERS

Arbitrarily sized integers, rationals and arbitrary precision floating point numbers are all stored in the class `numeric`. This is an interface that encapsulates the foundation class `cl_N` of Bruno Haible's C++ library `CLN` (Haible, 2000) in a completely transparent way.

[†]Strictly speaking, GiNaC does not have any abstract base classes in the C++ sense, since there are defaults for all methods. We therefore define an abstract base class to be one which does not make sense to instantiate.

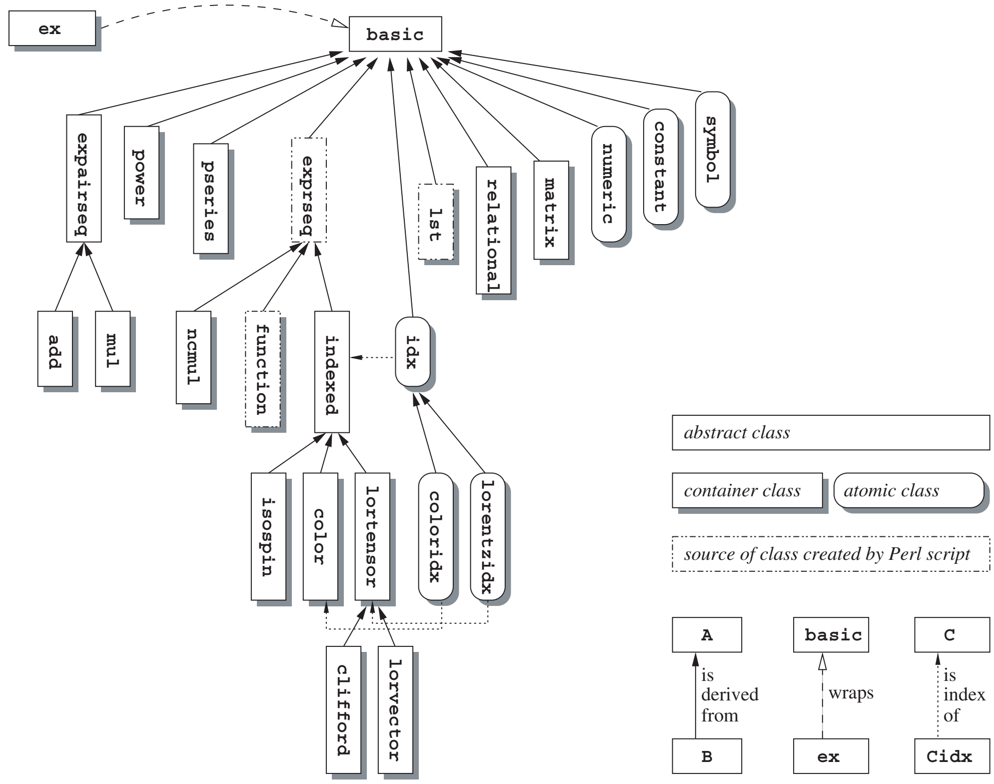

Figure 1. The GiNaC class hierarchy and some of the relations between the classes.

Table 1. List of the most important classes from Figure 1 and their purpose.

Class	Description	Examples
`symbol`	algebraic symbols	x
`numeric`	polymorphic CLN numbers	42, $\frac{7}{3}i$, 0.12345678
`constant`	symbols with associated `numeric`	π
`add`	sums of expressions	$a-2b+3$
`mul`	products of expressions	$2a^2(x+y+z)/b$
`power`	exponentials	x^2, $a^{(b+c)}$, $\sqrt{2}$
`pseries`	truncated power series	$x-\frac{1}{6}x^3+\mathcal{O}(x^5)$
`function`	symbolic functions	$\sin(2x)$
`lst`	list of expressions	$[x, 2y, 3+z]$
`relational`	relation between two expressions	x==y
`matrix`	matrices (and vectors) of expressions	$\begin{pmatrix} 1 & x \\ -x & 1 \end{pmatrix}$
`ncmul`	container for non-commutative objects	$\gamma_0\gamma_1$
`color`, `coloridx`	$SU(3)$ Lie algebra element, -index	T_a, δ_{ab}, f_{abc}
`lortensor`, `lorentzidx`	Lorentz tensor, -index	$p^\mu, g_{\mu\nu}$

The choice fell to `CLN` because it provides fast and asymptotically ideal algorithms for all basic operations (Karatsuba and Schönhage–Strassen multiplication) and a very flexible way of dealing with rationals and complex numbers. Also, it does not put any burden of memory management on us since all objects are reference-counted—just like GiNaC's—so there is no interference with garbage collection. Its polymorphic types are perfectly suited for implementing a CAS, and were indeed written with this intention. For instance, it honours the injection of the naturals into the rationals and of the complex numbers into the reals: rationals are instantaneously and efficiently normalized to coprime integer numerator and denominator and converted to integers if the resulting denominator is unity and complex numbers are instantaneously converted to reals if the imaginary part vanishes. Non-exact numbers i.e. floats and complex floats are constructed with any user-defined accuracy.

GiNaC provides functions and operators defined on class `numeric` to the user so the wrapper class `ex` may be circumvented. This provides some level of type-safety as well as a considerable speedup.

2.2. SYMBOLS AND CONSTANTS

Symbols are represented by objects of class `symbol`. Thus, construction of symbols is done by statements like `symbol x,y;`. In a compiled language like C++ the name of a variable is of course unavailable to the running program. For printing purposes, therefore, a constructor from a string is provided, i.e. `symbol x("x"),y("y");`. This is reminiscent of Common Lisp's (Steele, 1990) concept of *print name*. The responsibility for not mixing up names (as in `symbol x("y"),y("x");`) is entirely laid on the user. The string is not used at all for identification of objects. If omitted, the system will still deal out a unique string.

Unlike in other symbolic system evaluators, expressions may not be assigned to symbols. This is a restriction we had to introduce for the sake of consistency in the non-symbolic language C++. It is, however, possible to substitute a symbol within an expression with some other expression by calling the `.subs()` method.

Objects of class `constant` behave much like symbols except that they must return some specific number (if possible to arbitrary accuracy) when the method `.evalf()` is called. There are several predefined constants like π, etc. which have an associated function for numerical evaluation to arbitrary accuracy. Another possibility is an associated fixed precision `numeric`. Thus, physical constants are easily constructed by the user, as in this fragment:

```
constant qe("qe",numeric(1.60219e-19));  // elementary charge
cout << qe << endl;                      // prints 'qe'
cout << evalf(qe) << endl;               // prints '1.60219E-19'
```

2.3. POLYNOMIAL ARITHMETIC: THE CLASSES `add`, `mul` AND `power`

With the main object of interest being efficient multivariate polynomials and rational functions, GiNaC allows the creation of such objects using the overloaded operators `+`, `-`, `*` and `/` and the overloaded function `pow(`b`,`e`)` for exponentiation of expressions b and e.[†] When such an object is created, the built-in term rewriting rules of the classes `add` and `mul` are automatically invoked to bring it into a canonical form. Subsequent comparison of such objects is then easy and further supported by hash values. Due to the similarity

[†]It is also possible to overload operator `^` for exponentiation in C++, but this would lead to trouble since it always has lower precedence than `*`.

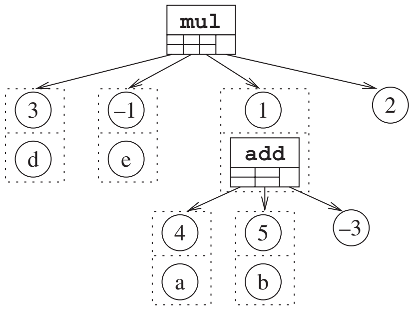

Figure 2. Internal representation of the multivariate rational function $2d^3(4a+5b-3)/e$.

in the rewriting rules for sums and products the actual implementation is mostly hidden in class `expairseq`, from which `add` and `mul` are derived. The internal representation is an unexpanded distributive one. For performance reasons numerical coefficients in front of monomials in sums and numerical exponents in products are treated separately (as shown in Figure 2). The term rewriting rules for class `power` are restricted to those simplifications that can be done efficiently.

GiNaC provides the usual set of operations on multivariate polynomials: determination of degree and coefficients, expansion of products over sums, collection of coefficients of like powers, conversion of rational expressions to a normal form (where numerator and denominator are relatively prime polynomials), decomposition of polynomials into unit part, content, and primitive part, and polynomial GCD and LCM computation. For the latter, GiNaC implements the heuristic polynomial GCD algorithm described in Liao and Fateman (1995), augmented by additional heuristics such as cancelling trivial common factors (e.g. x^n), eliminating variables that occur only in one polynomial, and special handling of partially factored polynomials. If the heuristic algorithm fails, GiNaC falls back to the subresultant PRS algorithm (Geddes *et al.*, 1992). This approach has so far proved successful for the application in `xloops`.

2.4. POWER SERIES: THE CLASS `pseries`

Expressions may be differentiated with respect to any symbol and also expanded as Taylor series or Laurent series. There is no distinction between those two. Series are internally stored in a truncated power series representation, optionally containing an order term, in a special class `pseries`. This class implements efficient addition, multiplication, and powering (including inversion) of series and can convert the internal representation to an ordinary GiNaC expression (polynomial) as well.

A program fragment where the mass increase from special relativity $\gamma=(1-(\frac{v}{c})^2)^{-1/2}$ is first Taylor expanded and then inverted and expanded again illuminates the behaviour and syntax of class `pseries` to some extent:

```
symbol v("v"), c("c");
ex gamma = 1/sqrt(1 - pow(v/c,2));
ex gamma_nr = gamma.series(v==0, 6);
cout << pow(gamma_nr,-2) << endl;
cout << pow(gamma_nr,-2).series(v==0, 6) << endl;
```

Raising the series $\gamma_{\rm nr}$ to the power -2 in line 4 just returns $\bigl(1 + \frac{1}{2}(\frac{v}{c})^2 + \frac{3}{8}(\frac{v}{c})^4 + \mathcal{O}(v^6)\bigr)^{-2}$. Only calling the series method again in line 5 makes the output simplify to $1 - v^2/c^2 + \mathcal{O}(v^6)$.

2.5. FUNCTIONS

C++ functions are not suited for symbolic expressions as arguments. This is so because, if the evaluation engine is unable to evaluate the argument, one wishes to return the function itself which would lead to an infinite recursion. If `x` is an indeterminate, then `sin(x)` is supposed to return `sin(x)`. In order to achieve this behaviour the class `function` is introduced. Each object of this class represents a single function (`sin`, `cos`…) and methods for evaluation, differentiation and so on may be attached to it. The C++ preprocessor is then used to define wrapper functions that return the corresponding objects of class `function`. This allows us to write functions down in C++ fashion and obtain the behaviour one knows from usual CASs:

```
symbol x("x"), y("y");
ex Do = Pi*(x+y/2);
cout << "sin(" << Do << ") -> " << sin(Do) << endl;
ex Re = Do.subs(y==1);
cout << "sin(" << Re << ") -> " << sin(Re) << endl;
ex Mi = Re.subs(x==11);
cout << "sin(" << Mi << ") -> " << sin(Mi) << endl;
ex Fa = Mi.evalf();
cout << "sin(" << Fa << ") -> " << sin(Fa) << endl;
```

The above fragment prints:

```
sin(Pi*(x+1/2*y)) -> sin(Pi*(x+1/2*y))
sin(Pi*(1/2+x)) -> sin(Pi*(1/2+x))
sin(23/2*Pi) -> -1
sin(36.128315516282622243) -> -1.0
```

A great many functions are already predefined in GiNaC, some of them, however, not yet with the full functionality. For instance, polygamma functions may not yet be evaluated numerically.

3. Benchmarks

Naturally, we want to know how GiNaC performs in comparison with other systems. Therefore we subject it and some other symbolic manipulators to several stress tests on different hardware architectures. All tests concentrate on non-C++ arithmetics (arbitrary precision instead of hardware-near `int`, `double`) and symbolic expressions. GiNaC is superior when it comes to algorithms that largely rely on machine-near data types. The first two tests were inspired by typical operation patterns in elementary particle physics where many different symbols and deeply nested functions need to be handled. They are designed to detect flaws in the memory management and the implementation of algorithms for manipulation of large container classes (products, sums…). This is done by having a close look at the asymptotic runtime behaviour.

The first test (Figure 3, left) consists of three steps:

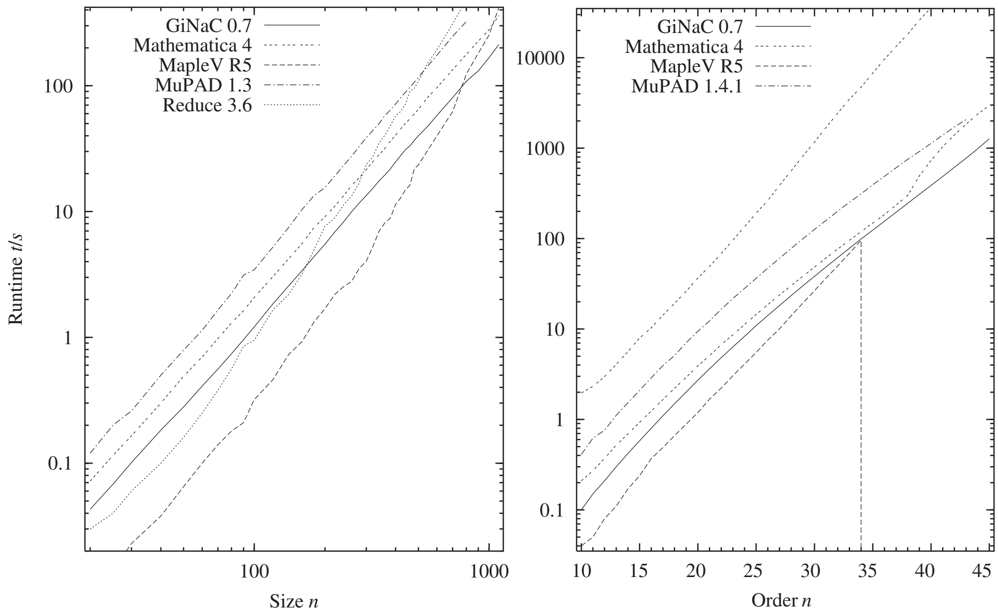

Figure 3. Runtimes for a substitute-expand consistency test (left) and for series expansion of $\Gamma(x)|_{x=0}$ (right). The tests are described in the text. The left graph was produced on an Alphaserver 8400, the right one on an Intel P-III.

(1) let e be the expanded sum of n symbols $\{a_0, \ldots a_{n-1}\}$ squared: $e \leftarrow \left(\sum_{i=0}^{n-1} a_i\right)^2$;
(2) in e substitute $a_0 \leftarrow -\sum_{i=2}^{n-1} a_i$;
(3) expand e again, it collapses to ${a_1}^2$.

The third step is the computationally expensive one. The system has to match terms in a sum of $\approx 2n^2$ elements and eliminate all but one. The timings are taken on an Alphaserver 8400 with CPUs of type EV5 running at 300 MHz under Digital Unix 4. This architecture was chosen specifically in order to give MapleV a chance, which has an internal limitation of $2^{16}-1$ terms in a sum on any other architecture.† This turns out to limit the test to $n < 182$. This also forced us to resort to a rather old version of MuPAD because no newer one is available for the Alpha platform. The tests were run until we got bored (which we defined to be 400 s). Further continuation would also have required more memory since some systems (particularly MuPAD) were allocating extraordinary amounts of RAM. The slopes of the curves are interesting: those systems that base their memory management on reference counts exhibit the quadratic scaling one would expect from the nature of the test while systems with a garbage collector (Maple and Reduce) start off faster and saturate earlier.

Next, we do a mixed test which, besides handling symbols, also involves the handling of large rational numbers and the evaluation of functions at certain points (Figure 3, right). We calculate the expansion of the gamma function around the pole at $x = 0$. The

†We do not have access to any newer version than MapleVR5. We were informed that the new release Maple6 does not suffer from the $2^{16}-1$ limitation any longer.

result up to order x^2 is:

$$\Gamma(x) = \frac{1}{x} - \gamma + \Big(\frac{\pi^2}{12} + \frac{\gamma^2}{2}\Big)x - \Big(\frac{\pi^2\gamma}{12} + \frac{\gamma^3}{6} + \frac{\zeta(3)}{3}\Big)x^2 + \cdots.$$

It is not completely clear what other systems are doing internally but GiNaC's implementation is simple and lacks any optimization. It falls back to the evaluation of polygamma functions $\psi_n(1)$ which in turn requires the evaluation of Riemann's Zeta function if their argument is even and, hence, to Bernoulli numbers. We show two curves for Mathematica, since this system decides to return the result in the form of unevaluated polygamma functions $\psi_n(1)$. If one insists on a result comparable with the other systems one is forced to introduce calls to `FunctionExpand[]`, which slows the system down more than an order of magnitude (upper curve). Without `FunctionExpand[]` its performance is only slightly worse than GiNaC's but with a funny excursion at high orders for which we do not have an explanation. It should, however, be mentioned that for Mathematica that ugly result in terms of $\psi_n(1)$ can, under certain circumstances, be acceptable since it may be handled further without resorting to ζ-functions. This becomes apparent when one tries to evaluate the coefficients in the resulting series numerically. Maple's internal limitation results in the breakdown at order $n = 35$ in this test.

Next, we apply GiNaC to a number of tests invented by Lewis and Wester (1999). Strictly speaking, these tests are very much geared towards the particular capabilities of the system Fermat. This explains the abundance of benchmarks on Smith and Hermite normal forms of matrices with numerical entries. Nevertheless, we tried to subject GiNaC to these tests where applicable.† The benchmarks were rerun on the same machine for those systems available to us and the rules of the game were slightly simplified: each system was given the chance to run as long as it needed but it was not allowed to allocate more than the physical memory available. The tests involving finite fields and the ones involving Smith and Hermite forms were skipped, since they are not applicable to GiNaC. Tests D and E were slightly rearranged in order to give a meaningful and comparable result: Maple and MuPAD were forced to cancel common factors in the result (using `normal`), something Pari-GP does automatically. The results shown in Table 2 are encouraging but show room for optimization. They also demonstrate some improvement of the other systems (notably Singular) over the original test performed by Lewis and Wester. The reader interested in a detailed description of the tests may consult Lewis and Wester (1999).

4. Conclusions and Further Work

Although the GiNaC framework was built specifically to become a symbolic engine for complex computations in quantum field theory it is our hope that it turns out to be useful for other applications, too. It provides only modest algebraic knowledge; instead it aims at being a fast and reliable foundation for combined symbolical/numerical/graphical projects in C++. It may be downloaded and distributed under the terms of the GNU general public license from `http://www.ginac.de/`. A tutorial introduction and complete cross references of the source code can also be found there.

Because the cycle edit-compile-execute common for all compiled languages may be rather tedious during development, care has been taken in the design of GiNaC to permit

†A fair number of these tests even found their way into the suite of GiNaC's regression tests.

Table 2. Runtimes in seconds for the tests proposed by Lewis and Wester (only as far as applicable to GiNaC) on an Intel P-III 450 MHz, 384 MB RAM running under Linux. Abbreviations used: GU (gave up), CR (crashed, out of memory), NA (not available), UN (unable, a prerequisite test failed).

	Benchmark	GiNaC 0.7	MapleV R5	MuPAD 1.4.1	Pari-GP 2.0.19β	Singular 1-3-7
A:	divide factorials $\frac{(1000+i)!}{(900+i)!}\Big\vert_{i=1}^{100}$	0.20	6.66	1.13	0.37	19.0
B:	$\sum_{i=1}^{1000} 1/i$	0.019	0.08	0.10	0.041	0.54
C:	gcd(big integers)	0.25	10.2	3.01	1.65	0.11
D:	$\sum_{i=1}^{10} iyt^i/(y+it)^i$	0.78	0.13	1.21	0.20	NA[a]
E:	$\sum_{i=1}^{10} iyt^i/(y+\vert 5-i\vert t)^i$	0.63	0.05	2.33	0.11	NA[a]
F:	gcd(2-var polys)	0.08	0.08	0.21	0.057	0.13
G:	gcd(3-var polys)	2.50	2.89	3.31	99.5	0.38
H:	det(rank 80 Hilbert)	10.0	33.5	42.5	3.97	CR
I:	invert rank 40 Hilbert	3.38	6.41	12.0	0.62	CR
J:	check rank 40 Hilbert	1.61	2.28	2.95	0.22	UN
K:	invert rank 70 Hilbert	22.1	92.0	74.0	5.90	CR
L:	check rank 70 Hilbert	9.19	21.6	14.2	1.57	UN
M_1:	rank 26 symbolic sparse, det	0.36	0.40	0.75	0.016	0.003
M_2:	rank 101 symbolic sparse, det	1903.3	GU	CR	CR	251.2
N:	eval poly at rational functions	CR	GU	CR	CR	NA
O_1:	three rank 15 dets (average)	43.2	GU	CR	CR	CR
O_2:	two GCDs	CR	UN	UN	UN	UN
P:	det(rank 101 numeric)	1.10	12.6	44.3	0.09	0.85
P':	det(less sparse rank 101)	6.07	13.3	46.2	0.38	1.25
Q:	charpoly(P)	103.9	1429.7	741.7	0.15	4.4
Q':	charpoly(P')	212.8	1497.3	243.1	CR	5.0

[a] We were informed that benchmarks D and E can indeed be performed with Singular—it is just not obvious what the right syntax is.

an interactive frontend to the library. Currently, there are two such interfaces. The first is the tiny GiNaC interactive shell `ginsh` for quickly manipulating some expressions. It does not provide any programming constructs, only back-reference to the last printed expressions. The second is an interface to the Cint C++ interpreter used extensively at CERN in the object-oriented data analysis framework ROOT (Brun and Rademakers, 1996).

Though at this stage GiNaC is already fully functional for the applications it was originally built for, numerous extensions are imaginable. The web page gives some hints in this direction and further suggestions are more than welcome, as are third-party contributions.

Acknowledgements

Part of this work was supported by 'Graduiertenkolleg Eichtheorien–Experimentelle Tests und theoretische Grundlagen' at University of Mainz. The authors wish to thank Oliver Welzel for fruitful discussions in the early phase of the project and Do Hoang Son for extensive testing. Stimulating comments came from Richard Fateman about efficiency in general, Michael Wester about his benchmarks, Stephen Watt about practical experiences with memory management schemes and from Dirk Kreimer and Hubert Spiesberger who contributed considerably by asking tons of good questions.

References

Brücher, L., Franzkowski, J., Kreimer, D. (1998). XLoops: Automated Feynman diagram calculation. *Comput. Phys. Commun.*, **115**, 140–160.

Brun, R., Rademakers, F. (1996). ROOT—an object oriented data analysis framework. In *Proceedings of AIHENP 97*. Lausanne, 1996, available from URL: `ftp://root.cern.ch/root/laussanne.ps.gz`.

Geddes, K. O., Czapor, S. R., Labahn, G. (1992). *Algorithms for Computer Algebra*. Boston, Kluwer Academic Publishers.

Haible, B. (2000). CLN, a class library for numbers. *see* URL: `http://clisp.cons.org/~haible/packages-cln.html`.

ISO/IEC 14882-1998(E) (1998). *Programming Languages—C++*. American National Standards Institute.

Lewis, R. H., Wester, M. (1999). Comparison of polynomial-oriented computer algebra systems. *SIGSAM Bulletin*, **33/4**, 5–13, available from URL: `http://www.fordham.edu/lewis/cacomp.html`.

Liao, H.-C., Fateman, R. (1995). Evaluation of the heuristic polynomial GCD. In *Proceedings of ISSAC 95*. Montreal, ACM Press, available from URL: `http://http.cs.berkeley.edu/~fateman/papers/phil8.ps`.

Steele, G. L. (1990). *Common Lisp the Language*, 2nd edn. Woburn, MA, Digital Press.

Vermaseren, J. A. M. (1991). *Symbolic Manipulation with Form, Version 2—Tutorial and Reference Manual*. Amsterdam, Computer Algebra Nederland.

Wester, M. ed. (1999). *Computer Algebra Systems: A Practical Guide*. Chichester, John Wiley & Sons.

Received 27 April 2000
Accepted 12 July 2001