What can it be used for

and

what can we learn from it?

Richard B. Kreckel, Johannes Gutenberg University, Mayence

Workshop on Open Source Computer Algebra Systems

Lyon, France, 21 May 2002

Overview

- The GiNaC Framework
  - Objective and Idea
  - The Class Hierarchy
- Demonstration: Selected Symbolic Capabilities
  - Manipulation of Polynomials
  - Power Series Expansion
- Symbolic Computation in C++: Relevant Design Patterns
  - Reference Counting
  - Anonymous Evaluation
  - Delegation vs. the Visitor Pattern
  - The Flyweight and the Bridge Patterns
  - Potential for Object Depletion

The GiNaC Framework: Objective and Idea

Why:

- Dissatisfaction with MapleV Rn for HEP computations:
  - Limitations in polynomial representation (fixed in 2000 with Maple6)
  - Many linguistical problems (weird scope)
  - Support by manufacturer? Patch releases? What's that?
- Situation in 1998: Nearly everything was closed-source or under 'strange' licenses (Form, Pari, NTL, Singular. . .) (different now!)
- C++ is *lingua franca* for us physicists
- Wanted a testbed for symbolic manipulation
- Could not afford frequent changes in language

The GiNaC Framework: Objective and Idea

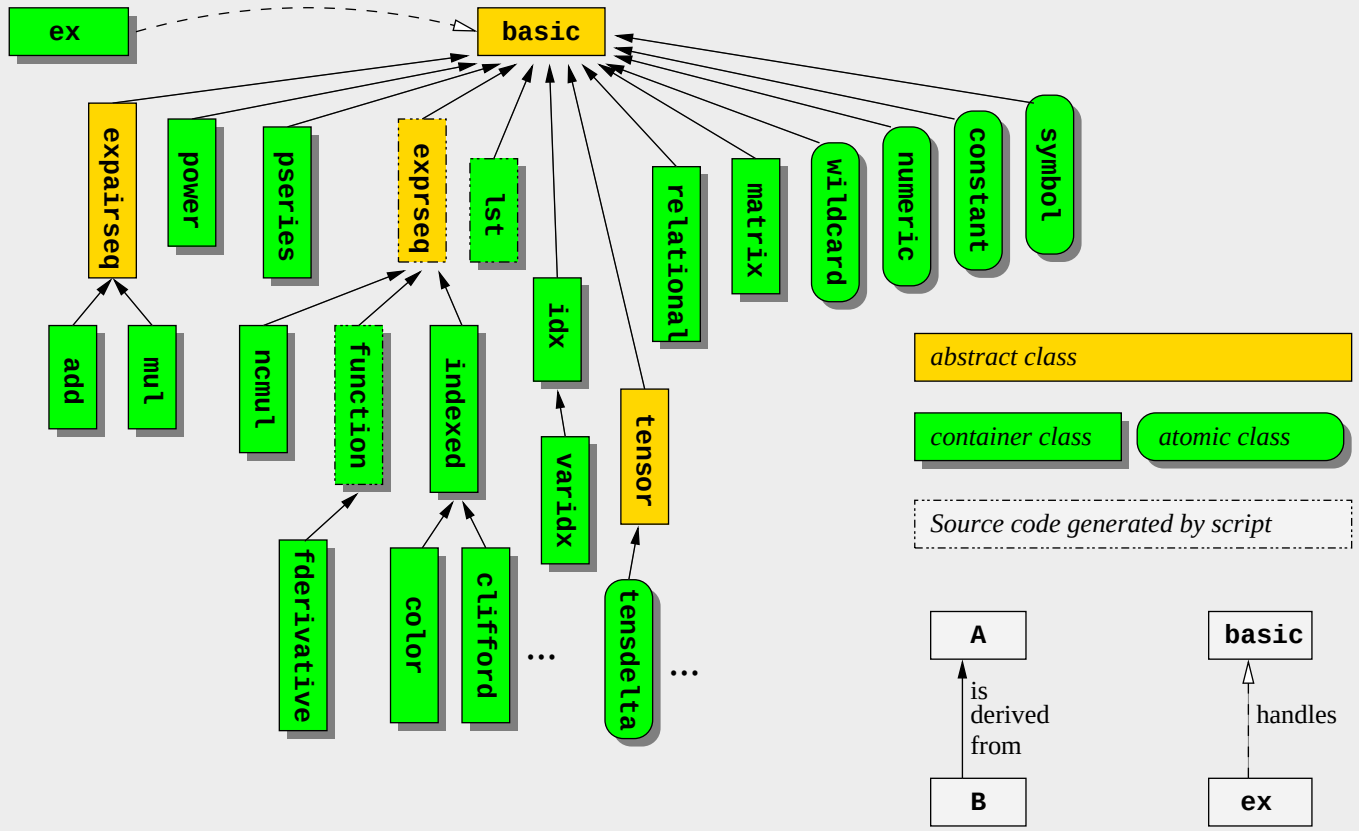
Why:

- Dissatisfaction with MapleV Rn for HEP computations:
  - Limitations in polynomial representation (fixed in 2000 with Maple6)
  - Many linguistical problems (weird scope)
  - Support by manufacturer? Patch releases? What's that?
- Situation in 1998: Nearly everything was closed-source or under 'strange' licenses (Form, Pari, NTL, Singular. . .) (different now!)
- C++ is *lingua franca* for us physicists
- Wanted a testbed for symbolic manipulation
- Could not afford frequent changes in language

GiNaC: (Ch. Bauer, A. Frink, R. Kreckel)

- Absolutely no limits (arithmetic, size of expressions)
- Free license (GPL), open development model (CVS, patches welcome)
- Not yet another language $\Rightarrow$ program in C++ (ISO/IEC 14882)
- Provides all that is needed for HEP computations:
  - Arbitrary precision arithmetic (through CLN)
  - Multivariate polynomial GCDs, though no factorization
  - Powerful series expansion
  - Noncommutative objects: SU(2), SU(3), etc. . .

The GiNaC Framework: The Class Hierarchy



- ▷ Memory management: Reference counting
- ▷ Objects are hashed, for fast comparison (Fibonacci hash)
- ▷ Class `numeric` wraps Bruno Haible's CLN (GPL, ∼100kloc)
  C++ library, arbitrary precision arithmetic, type retraction, refcounted
- ▷ Algebraic notation through operator overloading ⇒ intuitive syntax

Demonstration: Selected Symbolic Capabilities

Special Relativity (Einstein, 1905): mass m is a function of velocity v

$$m = \gamma m_0, \qquad \gamma = 1 \Big/ \sqrt{1 - \left(\frac{v}{c}\right)^2}$$

Demonstration: Selected Symbolic Capabilities

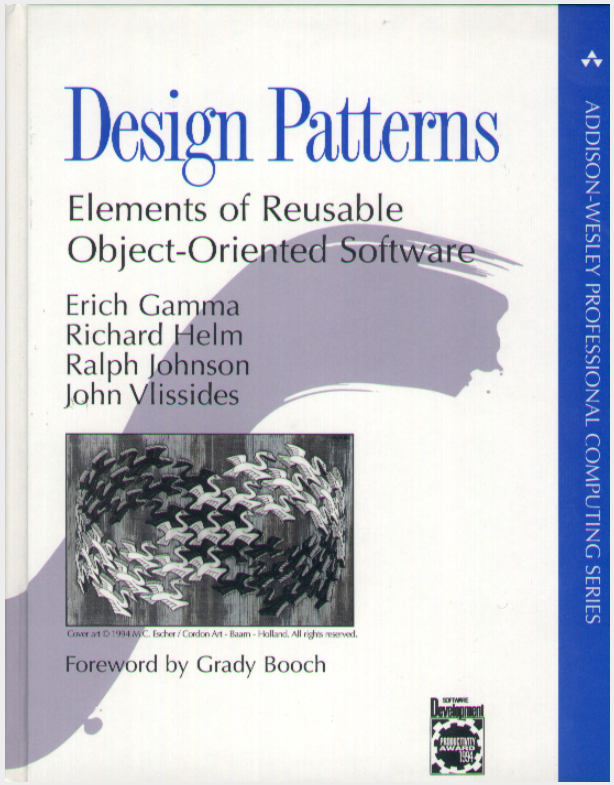
Special Relativity (Einstein, 1905): mass m is a function of velocity v

$$m = \gamma m_0, \qquad \gamma = 1 \Big/ \sqrt{1 - \left(\frac{v}{c}\right)^2}$$

```
1  #include <iostream>
2  #include <ginac/ginac.h>
3  using namespace std;
4  using namespace GiNaC;
5
6  int main(void)
7  {
8      const symbol v("v"), c("c");
9
10     ex gamma        = 1 / sqrt(1 - pow(v/c, 2));
11     ex mass_nonrel  = gamma.series(v == 0, 20);
12
13     cout << "the relativistic mass increase with v is\n"
14          << mass_nonrel << endl;
15
16     cout << "the inverse square of this series is\n"
17          << pow(mass_nonrel, -2).series(v == 0, 123) << endl;
18
19     return 0;
20 }
```

Symbolic Computation in C++: Relevant Design Patterns

Ad:



What is a Design Pattern?

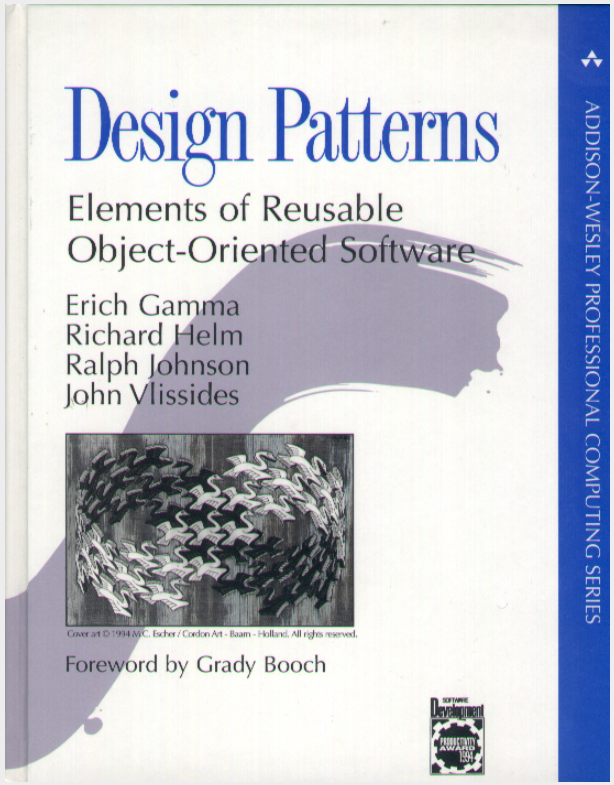
"*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*"

Christopher Alexander et al.
"A Pattern Language" (1977)

Design Patterns capture déjà-vus.

Symbolic Computation in C++: Relevant Design Patterns

Ad:



What is a Design Pattern?

"*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*"

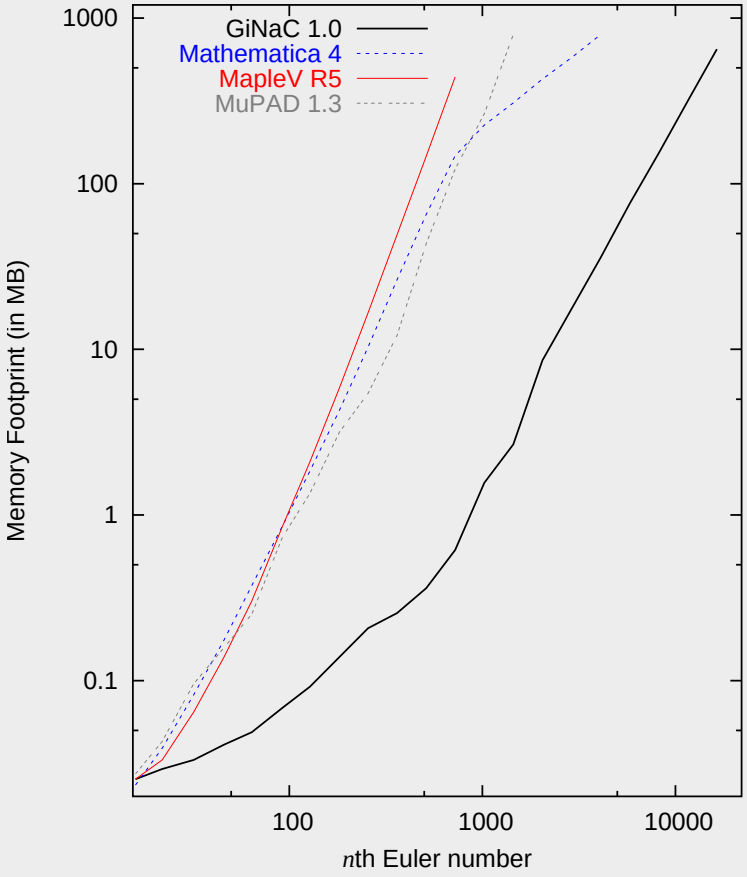
Christopher Alexander et al.
"A Pattern Language" (1977)

Design Patterns capture déjà-vus.

- Reference Counting
- Anonymous Evaluation
- Delegation vs. the Visitor Pattern
- The Flyweight and the Bridge Patterns
- Potential for Object Depletion

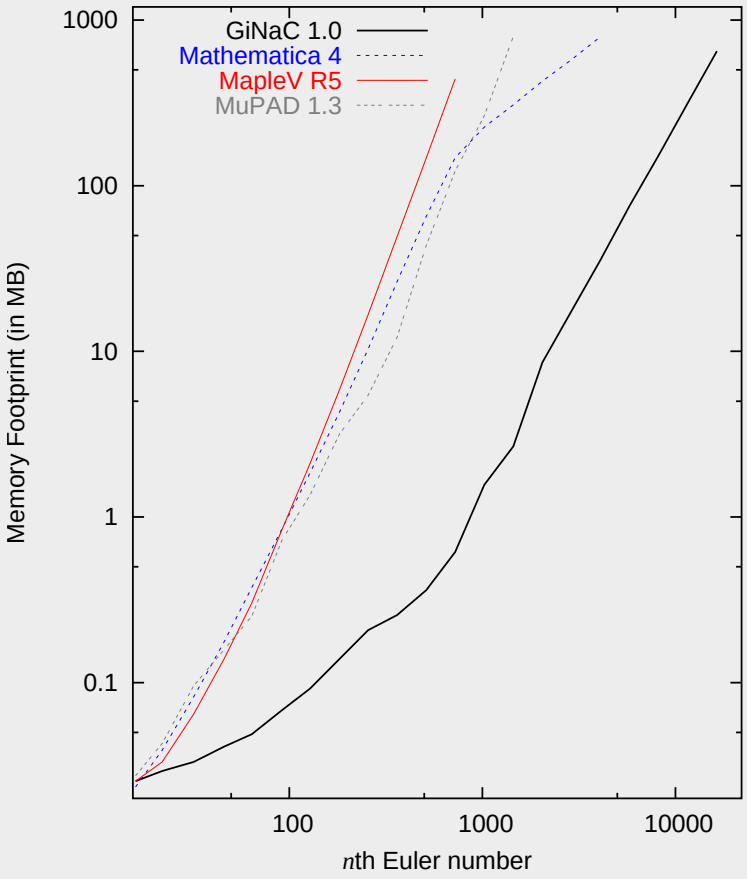
Relevant Design Patterns: Reference Counting

Euler Numbers $E_n :=$ Taylor coefficient in $\frac{1}{\cosh(x)} \equiv \sum_{n=0}^{\infty} E_n \frac{x^n}{n!}$:



Relevant Design Patterns: Reference Counting

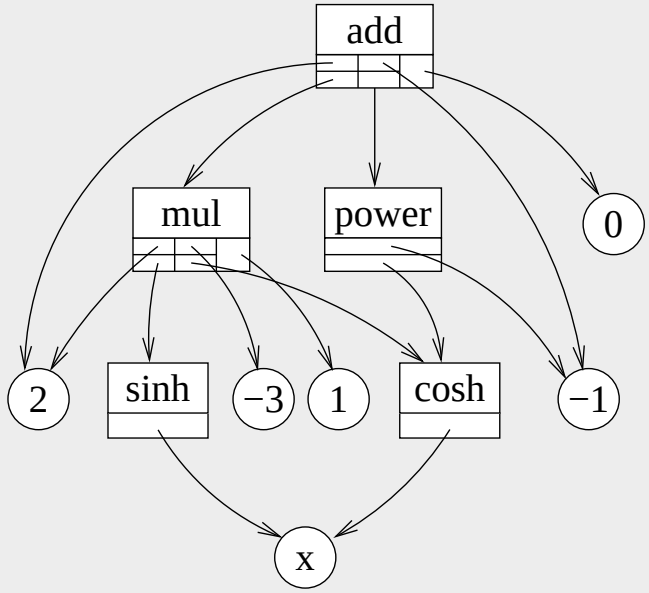
Euler Numbers E_n := Taylor coefficient in $\frac{1}{\cosh(x)} \equiv \sum_{n=0}^{\infty} E_n \frac{x^n}{n!}$:



GiNaC's representation tree for

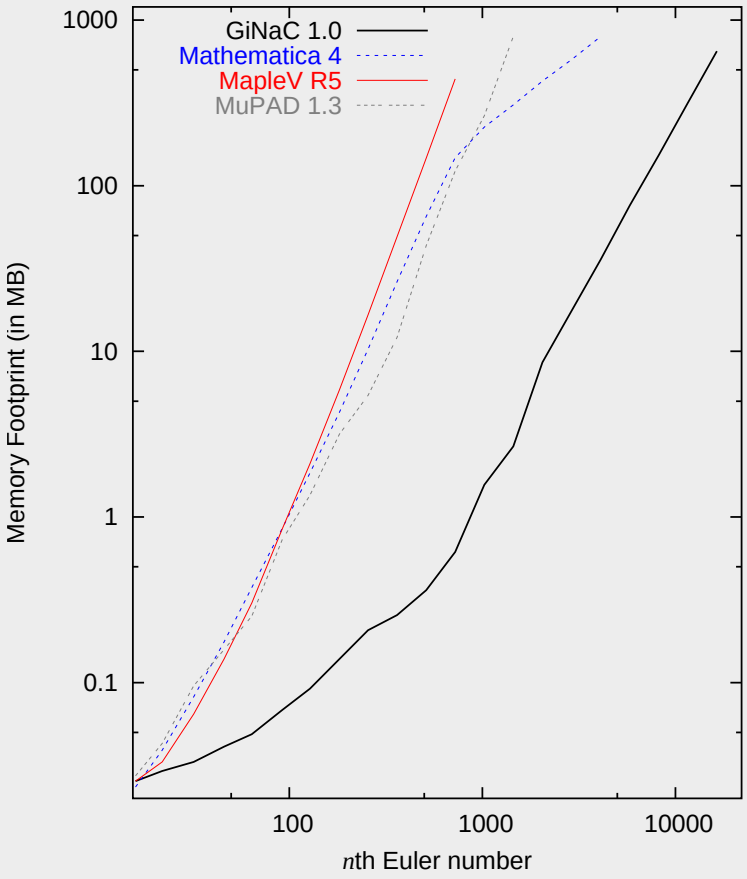
$$\frac{d^2}{dx^2}\frac{1}{\cosh(x)} = 2\frac{\sinh(x)^2}{\cosh(x)^3} - \frac{1}{\cosh(x)}$$

as it appears in E_2 looks like this:



Relevant Design Patterns: Reference Counting

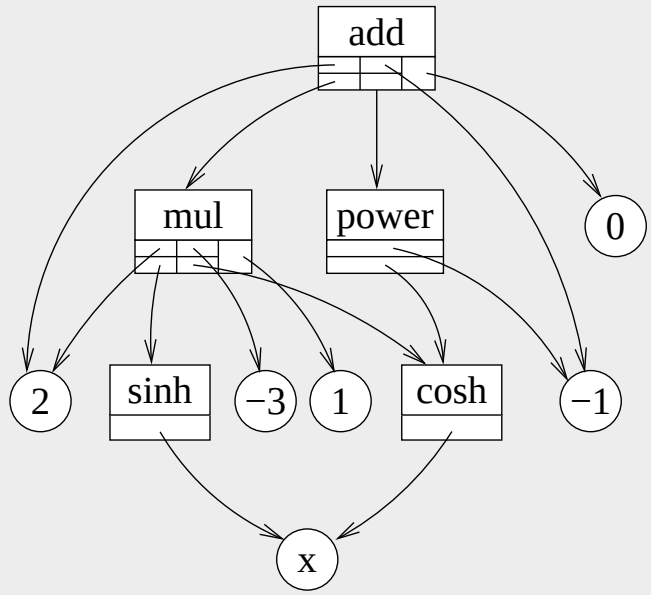
Euler Numbers E_n := Taylor coefficient in $\frac{1}{\cosh(x)} \equiv \sum_{n=0}^{\infty} E_n \frac{x^n}{n!}$:



GiNaC's representation tree for

$$\frac{d^2}{dx^2}\frac{1}{\cosh(x)} = 2\frac{\sinh(x)^2}{\cosh(x)^3} - \frac{1}{\cosh(x)}$$

as it appears in E_2 looks like this:



Tree ⇒ DAG

Relevant Design Patterns: Anonymous Evaluation

Problem: Need canonification of symbolic objects (sort of local CSE)

`x-x`	$\longrightarrow$	0	an integer
`x+x`	$\longrightarrow$	`2*x`	a product (or monomial)
`x+y`	$\longrightarrow$	`x+y`	a sum (or polynomial)

Relevant Design Patterns: Anonymous Evaluation

Problem: Need canonification of symbolic objects (sort of local CSE)

`x-x`	$\longrightarrow$	0	an integer
`x+x`	$\longrightarrow$	`2*x`	a product (or monomial)
`x+y`	$\longrightarrow$	`x+y`	a sum (or polynomial)

Solution: Implement method `::eval(int)` in each class, call it at the transition between ex and algebraic classes:

```
1  class ex {
2      // ...
3  private:
4      basic* bp;
5  };
```

```
1  ex::ex(const basic& other)
2  {
3      if (!(other.flags & status_flags::evaluated)) {
4          const ex& tmpex = other.eval(1); // evaluate only one (top) level
5          bp = tmpex.bp;
6          ++bp->refcount;
7          // clear up tmpex, etc...
8      }
9      // copy bp, adjust refcount, etc...
10 }
```

Relevant Design Patterns: Delegation vs. the Visitor Pattern

Common algorithms are best implemented as methods using 'Delegation'.

Method is added on each class, wrapper ex uses type dispatch:

```
1  inline const ex
2  ex::expand(void) const
3  {
4      return bp->expand();
5  }
```

```
1  const ex  // virtual
2  basic::expand(void) const
3  {
4      // maybe default...
5  }
```

```
1  const ex
2  add::expand(void) const
3  {
4      // overwrite...
5  }
```

etc. . .

Relevant Design Patterns: Delegation vs. the Visitor Pattern

Common algorithms are best implemented as methods using 'Delegation'.

Method is added on each class, wrapper ex uses type dispatch:

```
inline const ex
ex::expand(void) const
{
    return bp->expand();
}
```

```
const ex  // virtual
basic::expand(void) const
{
    // maybe default...
}
```

```
const ex
add::expand(void) const
{
    // overwrite...
}
```

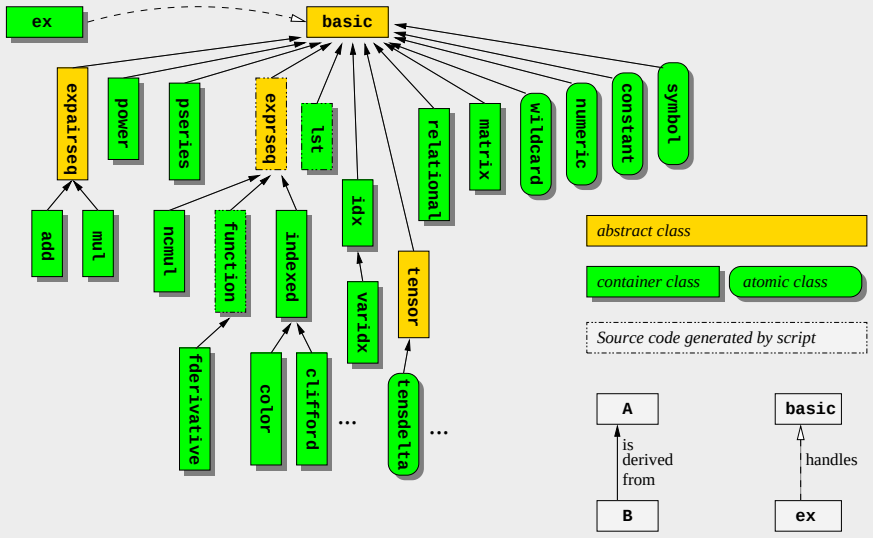
etc. . .

Problem: User-supplied algorithms would require adding methods, recompilation, breaking ABI, etc...
Solution: Let 'Visitor' object traverse the expression tree by reference.

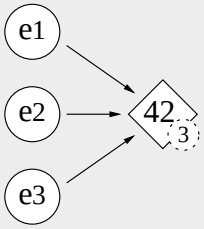
```
struct map_rem_quad
  : public map_function {
    ex var;
    map_rem_quad(const ex& var_)
      : var(var_) {}
    ex operator()(const ex& e)
    {
        if (is_a<add>(e) || is_a<mul>(e))
            return e.map(*this);
        else if (is_a<power>(e))
          /* return zero for even
           * powers of var_ */
        else
            return e;
    }
};

    // ex e constains x...
    map_rem_quad foo(x);
    cout << foo(e) << endl;
```

Relevant Design Patterns: The Flyweight and the Bridge Patterns

Why Flyweights matter:




```
1  ex e1 = 42;
2  ex e2 = e1;
3  ex e3 = 42ULL;
```

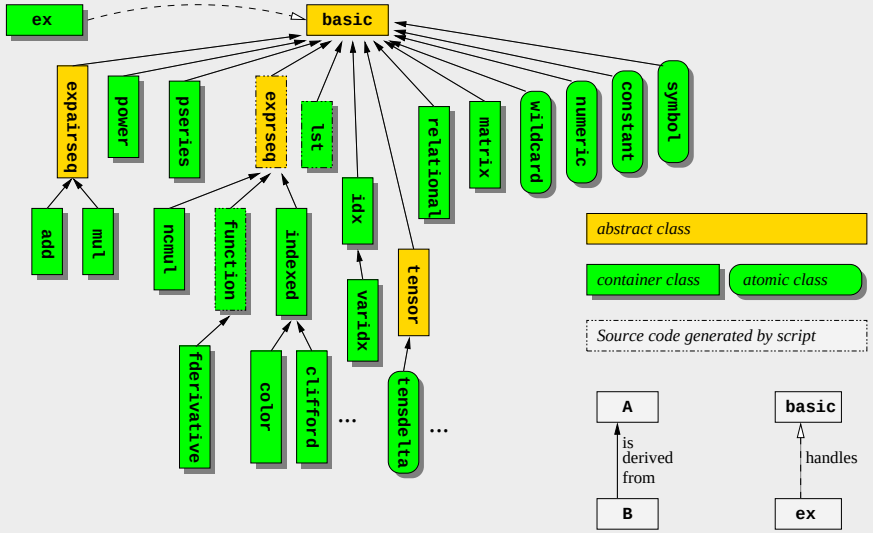


. . . and later:

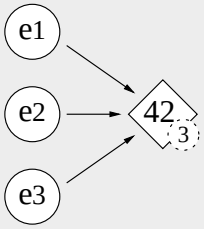
```
1  if (e1 == e2) {
2      // do something
```

Relevant Design Patterns: The Flyweight and the Bridge Patterns

Why Flyweights matter:



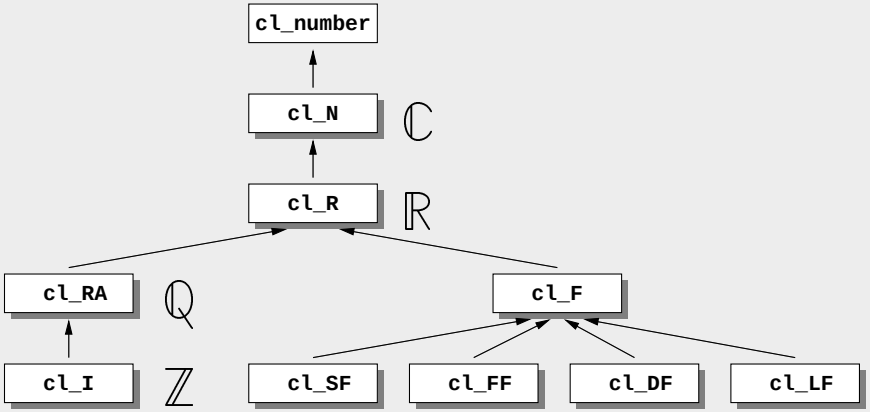

```
1  ex e1 = 42;
2  ex e2 = e1;
3  ex e3 = 42ULL;
```



. . . and later:

```
1  if (e1 == e2) {
2      // do something
```

c.f. CLN's class hierarchy:

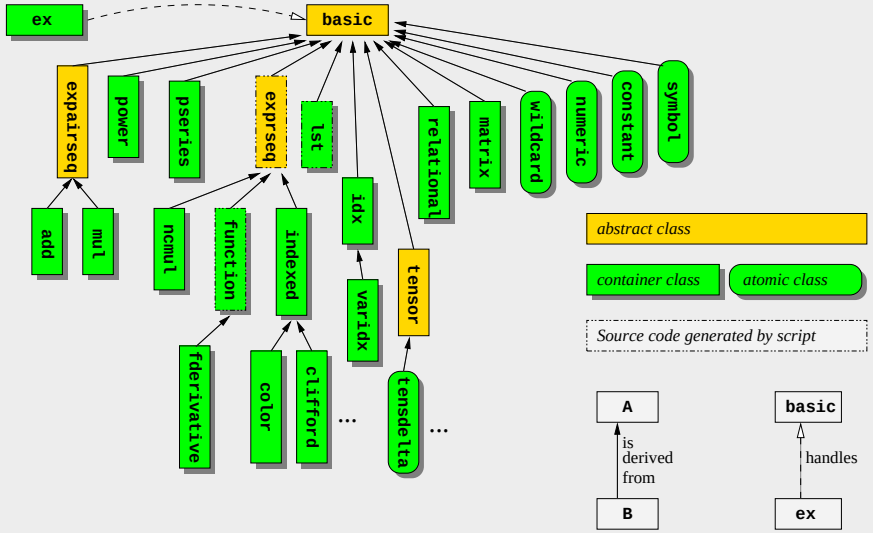


Positive: mathematically correct

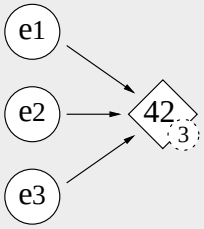
Negative: anti-OO

Relevant Design Patterns: The Flyweight and the Bridge Patterns

Why Flyweights matter:



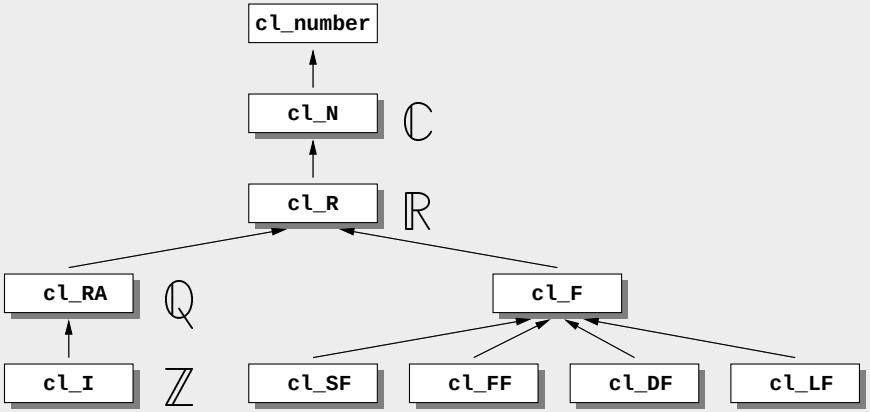

```
1  ex e1 = 42;
2  ex e2 = e1;
3  ex e3 = 42ULL;
```



. . . and later:

```
1  if (e1 == e2) {
2      // do something
```

c.f. CLN's class hierarchy:



Positive: mathematically correct

Negative: anti-OO

Solution: implementation must be completely hidden from people `#include`'ing `cl_I`, `cl_RA`, etc. ("Bridge")

Relevant Design Patterns: Potential for Object Depletion

Problem: Equivalent objects are created on the heap, calling for deep tree traversal whenever comparisons occur (i.e. very frequently!)

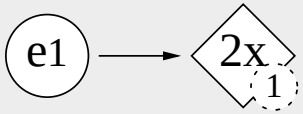
Relevant Design Patterns: Potential for Object Depletion

Problem: Equivalent objects are created on the heap, calling for deep tree traversal whenever comparisons occur (i.e. very frequently!)

Solution: Do it only once, and later compare pointers only!

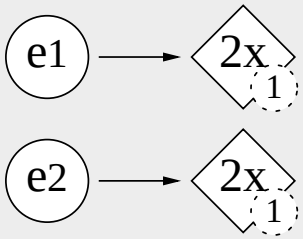
```
1   ex e1 = x+x;
```

declare e1 to store `x+x`
(canonifies to `2*x` on the heap)




```
2   ex e2 = 3*x-x;
```

declare e2 to store `3*x-x`
(canonifies to `2*x` on the heap)



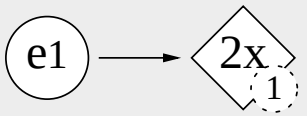
Relevant Design Patterns: Potential for Object Depletion

Problem: Equivalent objects are created on the heap, calling for deep tree traversal whenever comparisons occur (i.e. very frequently!)

Solution: Do it only once, and later compare pointers only!

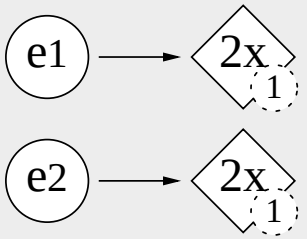
```
1   ex e1 = x+x;
```

declare e1 to store x+x
(canonifies to 2*x on the heap)



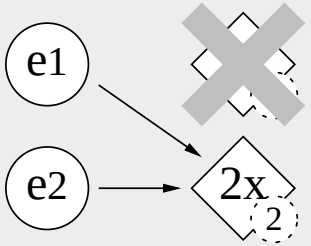

```
2   ex e2 = 3*x-x;
```

declare e2 to store 3*x-x
(canonifies to 2*x on the heap)




```
3   if (e1 == e2) {
4       //...
```

compare e1 with e2, since true, delete one of them and increase the refcount of the other one



Implementation is straightforward inside `operator==(ex,ex)`!

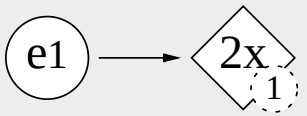
Relevant Design Patterns: Potential for Object Depletion

Problem: Equivalent objects are created on the heap, calling for deep tree traversal whenever comparisons occur (i.e. very frequently!)

Solution: Do it only once, and later compare pointers only!

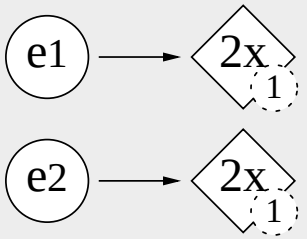
```
1   ex e1 = x+x;
```

declare e1 to store x+x
(canonifies to 2*x on the heap)



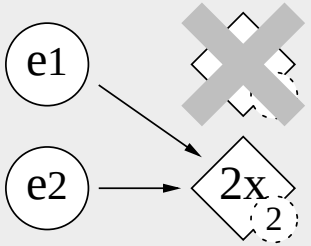

```
2   ex e2 = 3*x-x;
```

declare e2 to store 3*x-x
(canonifies to 2*x on the heap)




```
3   if (e1 == e2) {
4       //...
```

compare e1 with e2, since true, delete one of them and increase the refcount of the other one



Implementation is straightforward inside `operator==(ex,ex)`!

Prerequisite: Users must not be able to alias to reference-counted objects!
IOW: "Bridge" must be complete.

Availability, spin-offs and all that. . .

GiNaC 1.0.8 available from `http://www.ginac.de/`, focus on stability, distributed with 

,

and 

Availability, spin-offs and all that. . .

GiNaC 1.0.8 available from `http://www.ginac.de/`, focus on stability, distributed with 

, 

and 

gTybalt (Stefan Weinzierl, Roberta Marani) Cint, Root, TeXmacs
`http://www.fis.unipr.it/~stefanw/gtybalt.html`

pyginac (Pearu Peterson) Python bindings
`http://cens.ioc.ee/projects/pyginac/`

Symbolic Octave (Ben Sapp) Exposes GiNaC to GNU Octave
`http://bsoctave.sourceforge.net/`

Purrs (Roberto Bagnara et al.) Automated complexity analysis
`http://www.cs.unipr.it/purrs/`

Availability, spin-offs and all that. . .

GiNaC 1.0.8 available from `http://www.ginac.de/`, focus on stability, distributed with debian, SuSE and FreeBSD

gTybalt (Stefan Weinzierl, Roberta Marani) Cint, Root, TeXmacs
`http://www.fis.unipr.it/~stefanw/gtybalt.html`

pyginac (Pearu Peterson) Python bindings
`http://cens.ioc.ee/projects/pyginac/`

Symbolic Octave (Ben Sapp) Exposes GiNaC to GNU Octave
`http://bsoctave.sourceforge.net/`

Purrs (Roberto Bagnara et al.) Automated complexity analysis
`http://www.cs.unipr.it/purrs/`