CoCoALib

a C++ library for Computations in Commutative Algebra

John Abbott and Anna Maria Bigatti

Università di Genova, Italy

A guided tour in the CoCoA web page

What is CoCoA?

- **CoCoA** is a specialized computer algebra system.
- Its programming language is designed for non-programmers: no declarations, “natural mathematical syntax”, ...
- multilingual page: very simple examples of **CoCoA** syntax (please add more languages!)

Help system

- Reference card,
- (GUI) html help, pdf version (about 400 pages),
- Google search.

publications, download, conferences, CoCoASchool, **Redmine**...

Visit **http://cocoa.dima.unige.it**

CoCoA and CoCoALib

There are two main aspects in our project:

- **CoCoA** (5): the interactive system
- **CoCoALib**: the C++ source code

CoCoA is a beautiful car: you can drive it where you want...

... but if you need speed and power you need to be an expert driver and "talk" with the engine: **CoCoALib**.

CoCoALib has been designed to be an open source C++ library

in other words to be

used, **compiled**, and **extended** by everyone, not just the authors.

Design Philosophy behind CoCoA and CoCoALib

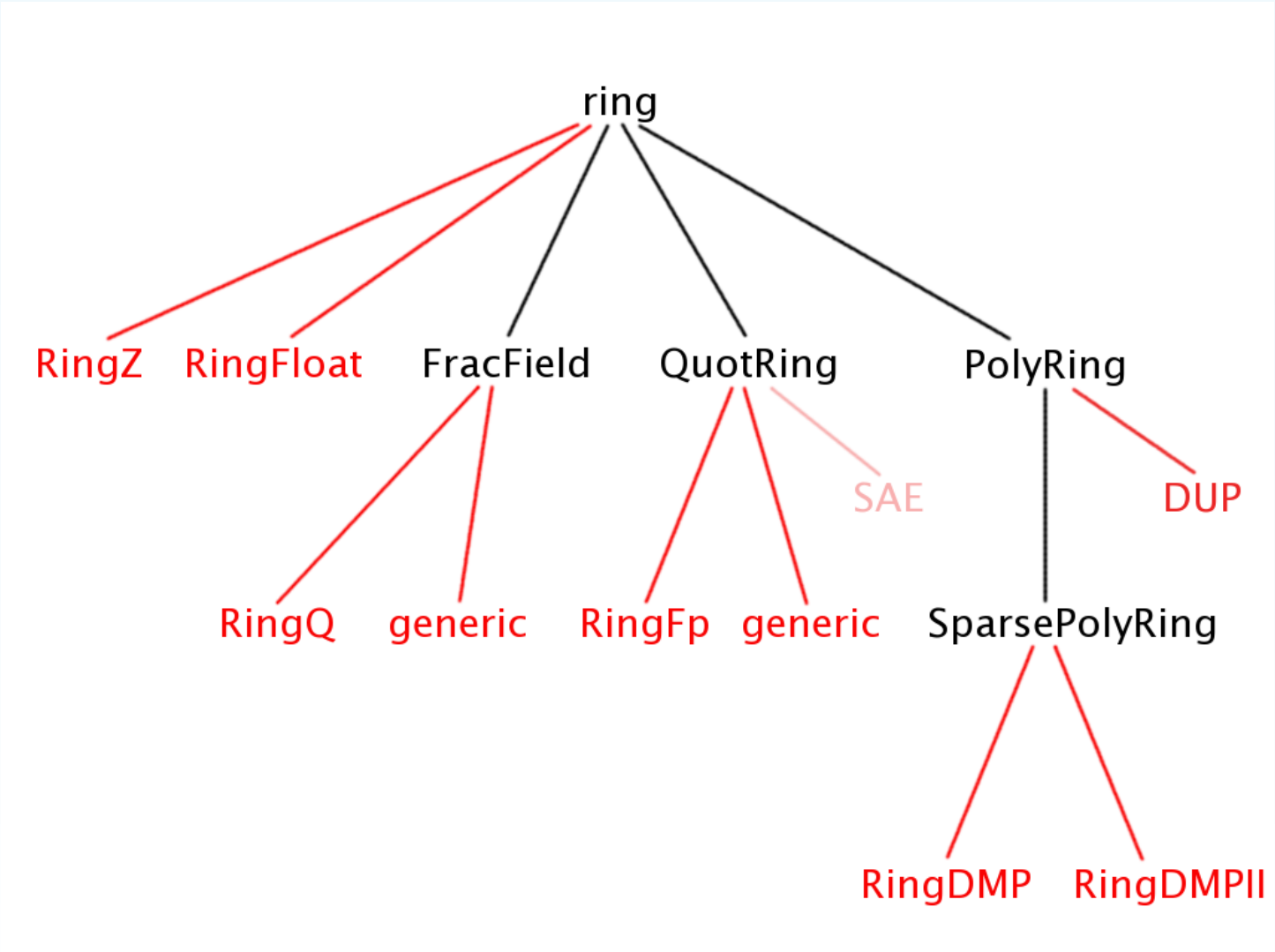
Basic goals of the design of **CoCoA** and **CoCoALib**:

- the code must be **easy and natural** to use
- the code must be **well documented** (for users & maintainers)
- the code must have **firm mathematical basis** (KR book)
- the code must be clear and **well designed**
- the code must be clean and **portable** (even on Windows ;-))
- the code must exhibit **good run-time performance**

Just a few obstacles moving from CoCoALanguage to C++

- we cannot write **2/3**: interpreted as integer division $\longrightarrow$ 0
  `BigRat(2,3)`
- we cannot write **x*y^4**: problems with operator priority $\longrightarrow$ `(x*y)^4`
  `x*power(y,4)`
- C++ lists/vectors are *a bit unfriendly* for C++ beginners

Ring Inheritance Diagram



The old and the new CoCoA: 4, Lib, Server, and 5

- **CoCoA** 0.99–1.5: 1997 (in Pascal)
- **CoCoA** 3–4: last version 4.7.5 (old and arthritic, in C)
- **CoCoALib** C++ library (young, spritely and flexible, **open source**)
- **CoCoAServer** is a prototype extensible server program;
  was used by **CoCoA-4** to call the features of **CoCoALib**.

- **CoCoA-5** system whose core is **CoCoALib**,
  with evolved language and capabilities.

God invented the integers...

Two ways to represent integers: **INT**/**BigInt** and **RingElem** $\in$ **ZZ**

```
BigInt three = BigInt(3);
BigInt seven = BigInt(7);
cout << seven/three;      // OK   value = 2
cout << -seven/three;     // OK   value = -3
cout << seven/(-three);   // OK   .......

RingElem three = RingElem(RingZZ(),3);
RingElem seven = RingElem(RingZZ(),7);
cout << seven/three;      // FAILS
cout << -seven/three;     // FAILS
cout << seven/(-three);   // FAILS
```

Two ways to represent rationals: **BigRat** and **RingQQ**

```
BigRat SevenThirds = BigRat(7,3);
cout << SevenThirds + 2/3;     // Nasty surprise!!
cout << SevenThirds + BigRat(2,3);// OK
```

Homomorphisms

```
ring ZZ = RingZZ();
ring P = NewPolyRing(ZZ, 3);   // P is ZZ[x[0..2]]

RingElem c(ZZ), f(P);
c = 3;   // same as  c = RingElem(ZZ,3);
f = 100; // same as  f = RingElem(P,100);

c * f;      // WRONG!  COCOA_ERROR!
c * LC(f); // in ZZ

RingHom phi = CanonicalHom(ZZ, P);
phi(c) * f;  // in P
```

(From **ex-RingHom2.C**)

Change of coordinates: **ex-RingHom3.C**

Coding conventions in CoCoA and CoCoALib

- single words: lower case **ideal**, **indet**, **coeff**, **ring**,..
- more words: *CamelCase* **RingElem**, **PolyRing**, **IdealOfPoints**, ...
- returning boolean: **Is** + *CamelCase* **IsEmpty**, **IsProbPrime**, **IsDivisible**, ..

- C++ member functions: **my** + *CamelCase* **myAdd**, **mySize**,...

CoCoALib with an analog C++ (STL) function:

push_back	**PushBack**
empty	**IsEmpty**

Some files in **CoCoALib** are called **Tmp...**: usually undocumented code, the operation is (or will become) official, but syntax might change.

Clean vs Efficient

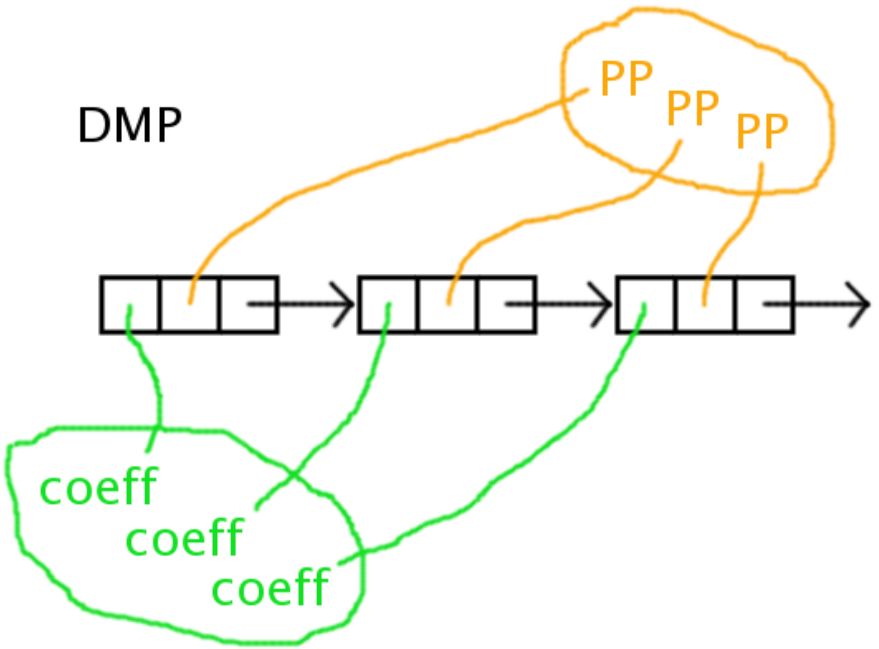
- **For all users** natural syntax with extensive checking
  `a = b + c;`
- **For experienced users** syntax for faster unchecked operations
  `R->myAdd(rawa, rawb, rawc);`
- **For developers** there are several debugging aids
  `MemPool`

General rule: use the **clean syntax**!

If you know how to **profile** (*e.g.* `gprof`) you will see how many times any function is called, and you then decide *if it is worth* using the faster dirtier call.

Polynomials

RingDistrMPoly, RingDistrMPolyInIPP, RingDistrMPolyInIFpPP



+ clean, easy to maintain, completely general

– poor locality, slow over F_q

DMPII (in some special cases)

+ good locality, fast

Power products (monomials/terms)

- **PPMonoidEv** exponent vector (also with BigInt exponents)
- **PPMonoidEvOv** exponent vector and order vector
- **PPMonoidOv** order vector (default for RingDistrMPolyInI(Fp)PP)
- **PPMonoidSparse** sparse representation
- **DynamicBitsets** omptimized squarefree power products (with many indeterminates)

(Run *ex-PPMonoidElem2.C*)

DivMask Implementation

Idea: define map $\phi : PP \to \{0,1\}^s$ from PPs to s-bitsets s.t.

$$t|t' \quad \Longrightarrow \quad \phi(t) \subseteq \phi(t')$$

Such ϕ are **DivMaskRule**s; many exist, none is universally best.

Example: $s = 32$ bits with DivMaskRule = **SingleBit**

PP:	x_0^2	x_1^0	x_2^0	x_3^5	x_4^0	x_5^3	x_6^1	x_7^0	x_8^3	x_9^1	...
Bitset:	1	0	0	1	0	1	1	0	1	1	...

C++ Inheritance: user can choose DivMaskRule at run-time, so computing a DivMask is "slow", but subset test is the same for all rules $\Longrightarrow$ inline $\Longrightarrow$ fast.

Twin Float Arithmetic: floats you can trust

Each value is represented as a pair of high-precision floats, and both components must have approximately the same value. Based on idea in Traverso & Zanoni, ISSAC 2002.

Colour key: **Precision requested; guard digits; trouble; noise.**

"Safe" value	$\left\{\begin{array}{l}1.0000000000000005357\\1.0000000000000001079\end{array}\right\}$
Noise just acceptable	$\left\{\begin{array}{l}1.0000000000003141592\\1.0000000000014142135\end{array}\right\}$
Noise unacceptable	$\left\{\begin{array}{l}1.0000000000031415926\\1.0000000000141421356\end{array}\right\}$ $\Rightarrow$ ERROR insuff. prec.

1. The **green** and **blue** digits must always match.
2. We **trust only the green** digits to be correct.

(Run *examples/ex-RingTwinFloat2.C*)

How to join in

Prerequisites

- Some knowledge of basic C++ programming
- Mild familiarity with compilation
- the GMP library (and Boost library, for some operations)

– Visit **CoCoA web page**

What to do

- Download the latest **CoCoALib** from
  `http://cocoa.dima.unige.it/cocoalib/`
- Configure and compile
  `./configure; make`
- Play and experiment!
  `cd examples; make`

– Compile and run (Run *ex-empty.C*)

Documentation

- text (t2t), pdf, html: corresponding to the main classes
- `examples/` directory:
  - focus on a class and give all its functions e.g. `ex-RingElem1.C`
  - "pieces of code" explaining particular functions, e.g. `ex-PolyRing1.C`
  - workarounds for missing or incomplete aspects, e.g. `ex-AlexanderDual.C`

Adding a function to the CoCoA-5

- Experiment in `ex-empty.C`
- Add the function to **CoCoALib**
  - Make new files `AnnaFile.H` and `AnnaFile.C` with your C++ code
  - Save `AnnaFile.H` in `include/CoCoA/` (and run `make` in `include/CoCoA/`)
  - Save `AnnaFile.C` in `src/AlgebraicCore/` (and add it to `Makefile`)
- Add the function call in `src/CoCoA-5/BuiltInFunctions.C` or `src/CoCoA-5/BuiltInOneLiners.C`