Evaluation of a Java Computer Algebra System

Heinz Kredel

ASCM 2007, Singapore

ASCM

Introduction

- object oriented design of a computer algebra system

  = software collection for symbolic (non-numeric) computations
- type safe through Java generic types
- thread safe, ready for multicore CPUs
- dynamic memory system with GC
- 64-bit ready
- jython (Java Python) front end

ASCM

Overview

- Introduction
- Example
- Types introduction
- Evaluation
- Conclusions

Chebychev polynomials

defined by recursion:

```
T[0] = 1
T[1] = x
T[n] = 2 x T[n-1] - T[n-2]
```

first 10 polynomials:

```
T[0] = 1
T[1] = x
T[2] = 2 x^2 - 1
T[3] = 4 x^3 - 3 x
T[4] = 8 x^4 - 8 x^2 + 1
T[5] = 16 x^5 - 20 x^3 + 5 x
T[6] = 32 x^6 - 48 x^4 + 18 x^2 - 1
T[7] = 64 x^7 - 112 x^5 + 56 x^3 - 7 x
T[8] = 128 x^8 - 256 x^6 + 160 x^4 - 32 x^2 + 1
T[9] = 256 x^9 - 576 x^7 + 432 x^5 - 120 x^3 + 9 x
```

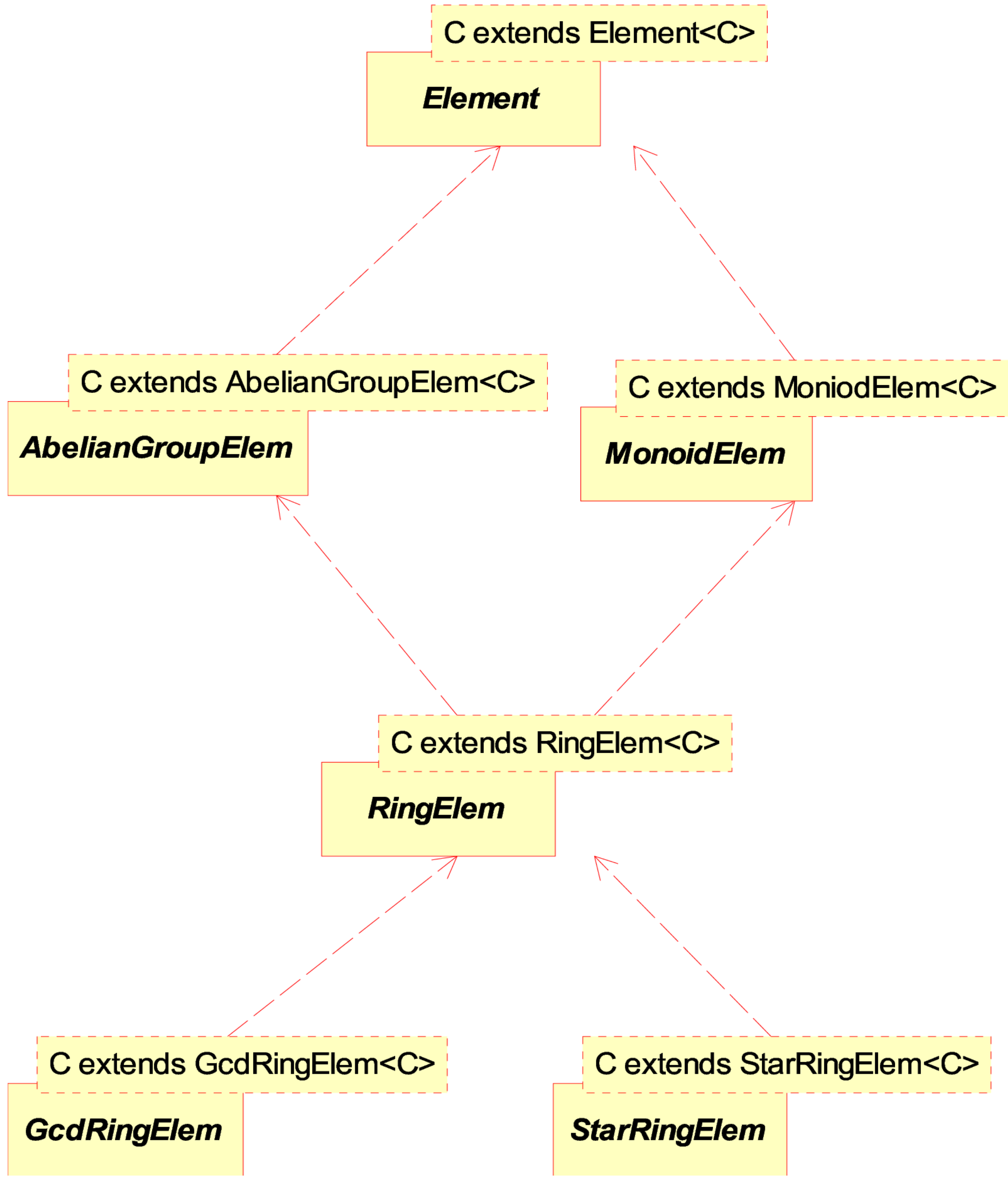
ASCM

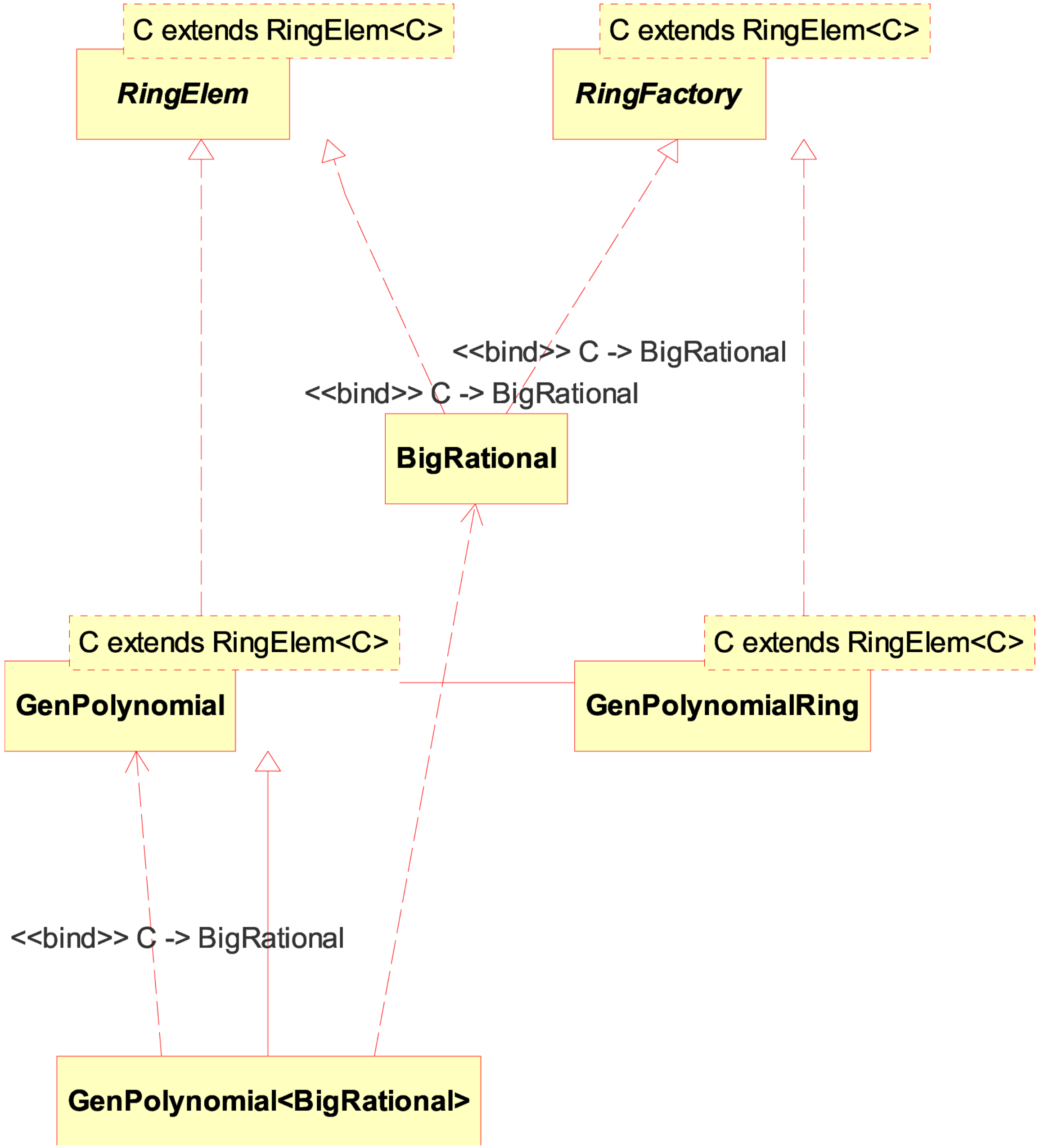
Chebychev polynomial computation

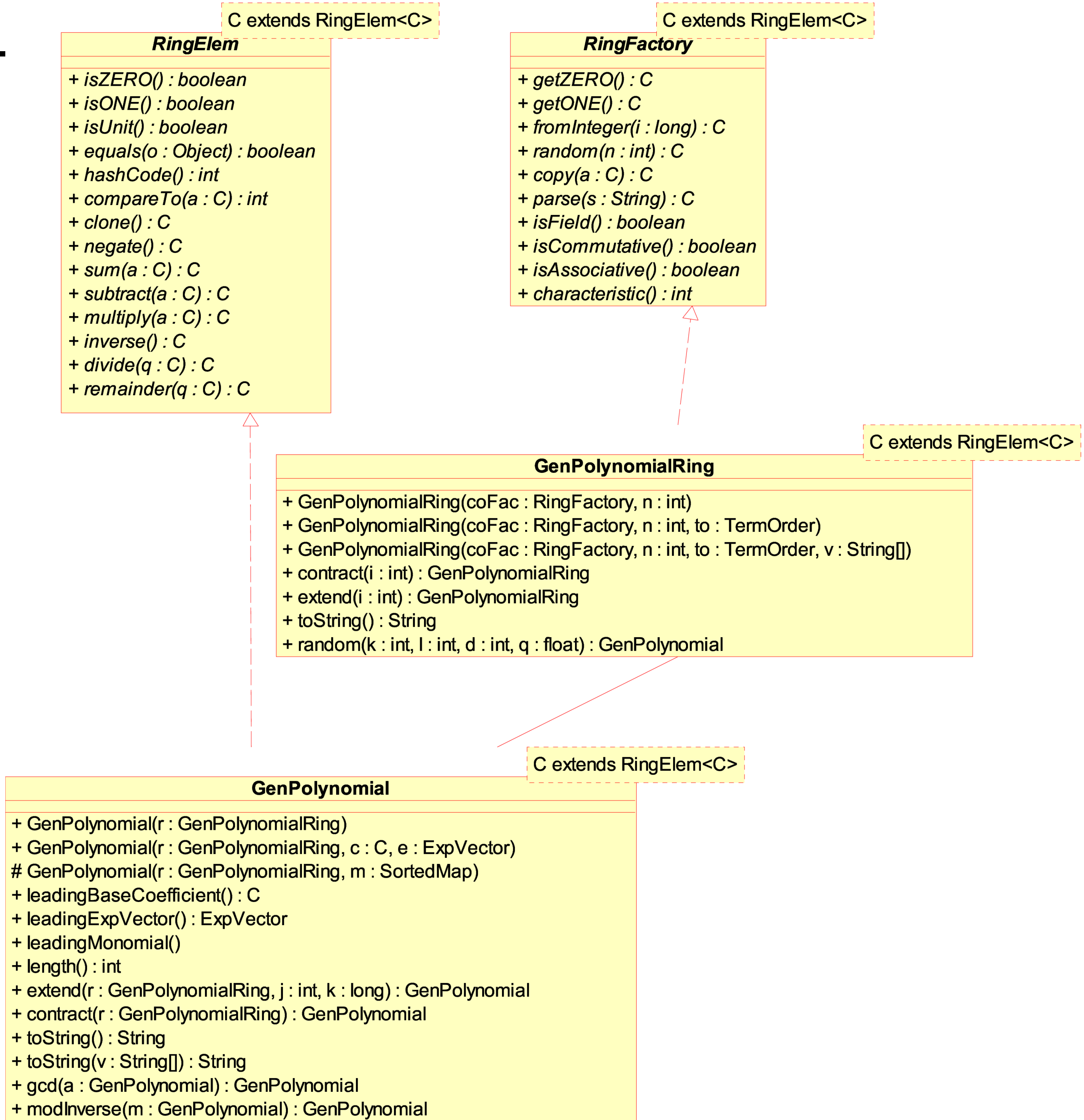
```
int m = 10;
BigInteger fac = new BigInteger();
String[] var   = new String[]{ "x" };
GenPolynomialRing<BigInteger> ring
                = new GenPolynomialRing<BigInteger>(fac,1,var);
List<GenPolynomial<BigInteger>> T
                = new ArrayList<GenPolynomial<BigInteger>>(m);
GenPolynomial<BigInteger> t, one, x, x2;
one = ring.getONE();
x   = ring.univariate(0); // polynomial in variable 0
x2  = ring.parse("2 x");
T.add( one );     // T[0]
T.add( x );       // T[1]
for ( int n = 2; n < m; n++ ) {
    t = x2.multiply( T.get(n-1) ).subtract( T.get(n-2) );
    T.add( t );   // T[n]
}
for ( int n = 0; n < m; n++ ) {
    System.out.println("T["+n+"] = " + T.get(n) );
}
```

Overview

- Introduction
- Example
- Types introduction
- Evaluation
- Conclusions


C extends Element<C>
Element
C extends AbelianGroupElem<C>
AbelianGroupElem
C extends MoniodElem<C>
MonoidElem
C extends RingElem<C>
RingElem
C extends GcdRingElem<C>
GcdRingElem
C extends StarRingElem<C>
StarRingElem


C extends RingElem<C>
RingElem
C extends RingElem<C>
RingFactory
<<bind>> C -> BigRational
<<bind>> C -> BigRational
BigRational
C extends RingElem<C>
GenPolynomial
C extends RingElem<C>
GenPolynomialRing
<<bind>> C -> BigRational
GenPolynomial<BigRational>


C extends RingElem<C>
RingElem
+ isZERO() : boolean
+ isONE() : boolean
+ isUnit() : boolean
+ equals(o : Object) : boolean
+ hashCode() : int
+ compareTo(a : C) : int
+ clone() : C
+ negate() : C
+ sum(a : C) : C
+ subtract(a : C) : C
+ multiply(a : C) : C
+ inverse() : C
+ divide(q : C) : C
+ remainder(q : C) : C
C extends RingElem<C>
RingFactory
+ getZERO() : C
+ getONE() : C
+ fromInteger(i : long) : C
+ random(n : int) : C
+ copy(a : C) : C
+ parse(s : String) : C
+ isField() : boolean
+ isCommutative() : boolean
+ isAssociative() : boolean
+ characteristic() : int
C extends RingElem<C>
GenPolynomialRing
+ GenPolynomialRing(coFac : RingFactory, n : int)
+ GenPolynomialRing(coFac : RingFactory, n : int, to : TermOrder)
+ GenPolynomialRing(coFac : RingFactory, n : int, to : TermOrder, v : String[])
+ contract(i : int) : GenPolynomialRing
+ extend(i : int) : GenPolynomialRing
+ toString() : String
+ random(k : int, l : int, d : int, q : float) : GenPolynomial
C extends RingElem<C>
GenPolynomial
+ GenPolynomial(r : GenPolynomialRing)
+ GenPolynomial(r : GenPolynomialRing, c : C, e : ExpVector)
# GenPolynomial(r : GenPolynomialRing, m : SortedMap)
+ leadingBaseCoefficient() : C
+ leadingExpVector() : ExpVector
+ leadingMonomial()
+ length() : int
+ extend(r : GenPolynomialRing, j : int, k : long) : GenPolynomial
+ contract(r : GenPolynomialRing) : GenPolynomial
+ toString() : String
+ toString(v : String[]) : String
+ gcd(a : GenPolynomial) : GenPolynomial
+ modInverse(m : GenPolynomial) : GenPolynomial

Implementation

- 140 classes and interfaces
- plus 70 JUnit test cases
- JDK 1.5 with generic types
- javadoc API documentation
- logging with Apache Log4j
- build tool is Apache Ant
- revision control with subversion
- some jython (Java Python) scripts
- open source, license is GPL or LGPL

ASCM

Overview

- Introduction
- Example
- Types introduction
- Evaluation
- Conclusions

Interfaces as types

- CAS in C++ not possible since no interfaces, (multiple) inheritance is not sufficient [28,29]
- need separate abstract type structure for interfaces and implementations
- have interfaces and classes in Java
- Axiom/Aldor: categories and domains [6,7]
- SmallTalk: views and classes with free renaming [30]
- Java: facade pattern to map names at runtime
- “Problem”: `GenSolvablePolynomial<C> extends GenPolynomial<C> implements RingElem<GenSolvablePolynomial<C>>`

ASCM

Generics and inheritance

- generics in Java since JDK 1.5
- generics can be simulated by a well-designed type hierarchy [32]
- before [31]: `Coefficient` and `Polynomial`
- but generics bring more type safety
- now cannot multiply polynomials with `BigInteger` and `BigRational` coefficients
- clear type denotation: `List<GenPolynomial< AlgebraicNumber<ModInteger>>>`

Dependent types

- polynomials in different number of variables have same type
- finite rings and fields have same type
- also term order is not denoted in the type
- SmallTalk types are first class objects:
  - `class Mod7 = ModIntegerRing(7);`
  - `Mod7 x = new Mod7(1);`
- `GenPolynomialRing<BigInteger,Var5>`
- other systems use coercion [19]
- carves hole in our type system

Method semantics

- methods with undefined semantics in some rings
  - what is `signum()` in unordered rings?
  - `divide(), remainder()` only for non-zero divisor, of limited value for multivariate polynomials
  - `inverse()` may fail if element is not invertible in ring
- Axiom/Aldor returned “failed” type
- we allow any meaningful reaction:
  - return predefined value
  - throw checked exception or unchecked run-time exception
- test methods `isZERO(), isUnit(), isField()`

Recursive types

- needed in greatest common divisor algorithms
- `RingElem<C extends RingElem<C>>`
- `GenPolynomial<GenPolynomial<ModInteger>>`
- raw type is `GenPolynomial`
- so can't overload and need to duplicate code
    - `baseGcd( GenPolynomial<C> a, b )`
    - `recursiveGcd( GenPolynomial<GenPolynomial<C>> a, b)`
- implemented abstract GCD class and specific
    - polynomial remainder sequences (PRS)
    - and modular methods with chinese remaindering

Factory pattern

- how to create 0, 1, polynomial in x or random elements i polynomial rings?
  - need a way to create respective coefficients
- idea: use factory pattern for all element creations
  - polynomial factories have factories for coefficients
- also applied in `GCDFactory` to select appropriate PRS oder modular implementation

```
GreatestCommonDivisor<BigInteger> engine =
  GCDFactory.<BigInteger>getImplementation(coFac);
c = engine.gcd(a,b);
```

  - others [24,25]: requirement oriented programming

ASCM

Code reuse (1)

- SAC-2/Aldes [14] and MAS [12]
  - three polynomial representations
  - with three or more coefficient implementations
  - e.g. `IPPROD, DIRPPR, DMPPRD`
- arbitrary domain system of MAS
  - 13 implemented coefficients selectable at run-time
  - with 20% performance penalty and limited type safety
- now in JAS
  - only one representation (is questionable [16,17])
  - but works for all 10+ coefficient implementations

Code reuse (2)

- using (object oriented) inheritance
  - abstract Groebner base class with sequential or parallel implementations
  - abstract greatest common divisor class with PRS and modular implementations
- maximum code reuse in e-Groebner base [26] implementation

```
public class EGroebnerBaseSeq<C extends RingElem<C>>
  extends DGroebnerBaseSeq<C> {

  public EGroebnerBaseSeq(EReductionSeq<C> red){ . }

/* nothing to implement */ }
```

Performance

- polynomial arithmetic performance:
  - performance of coefficient arithmetic
    - `java.math.BigInteger` in pure Java, faster than GMP style JNI C version
  - sorted map implementation
    - from Java collection classes with known efficient algorithms
  - exponent vector implementation
    - using `long[]`, have to consider also `int[]` or `short[]`
    - want `ExpVector<C>` but generic types may not be elementary types
  - JAS comparable to general purpose CA systems but slower than specialized systems

Performance

$[37]\,compute\; q = p \times (p+1)$

$$p = (1+x+y+z)^{20}$$
$$p = (10000000001\,(1+x+y+z))^{20}$$
$$p = (1+x^{2147483647}+y^{2147483647}+z^{2147483647})^{20}$$

JAS: options, system	JDK 1.5	JDK 1.6
BigInteger, G	16.2	13.5
BigInteger, L	12.9	10.8
BigRational, L, s	9.9	9.0
BigInteger, L, s	9.2	**8.4**
BigInteger, L, big e, s	9.2	**8.4**
BigInteger, L, big c	66.0	59.8
BigInteger, L, big c, s	45.0	**43.2**

options, system	time	@2.7GHz
MAS 1.00a, L, GC = 3.9	**33.2**	
Singular 2-0-6, G	2.5	
Singular, L	**2.2**	
Singular, G, big c	12.9	
Singular, L, big exp	out of memory	
Maple 9.5	**15.2**	9.1
Maple 9.5, big e	19.8	11.8
Maple 9.5, big c	64.0	38.0
Mathematica 5.2	**22.8**	13.6
Mathematica 5.2, big e	30.9	18.4
Mathematica 5.2, big c	30.6	18.2
JAS, s	8.4	5.0
JAS, big e, s	8.6	5.1
JAS, big c, s	47.8	28.5

ASCM

Computing times in seconds on AMD 1.6 GHz or 2.7 GHz Intel XEON CPU.
Options are: coefficient type, term order: G = graded, L = lexicographic,
big c = using the big coefficients, big e = using the big exponents, s = server JVM.

Applications

- polynomial reduction
- Buchbergers algorithm to compute Groebner bases
- not much (mathematical) optimization yet, simple structure used also for parallel implementation
- sequential, parallel and distributed versions
- non-commutative left, right and two-sided versions
- modules over polynomial rings and syzygies
- greatest common divisors
- d- and e-Groebner bases

Parallelization (1)

- thread safety from the beginning
  - explicit synchronization
  - immutable algebraic objects
- utility classes now from `java.util.concurrent`
- parallel proxy for greatest common divisor
  - `GreatestCommonDivisor<BigInteger> engine = GCDFactory.<BigInteger>getProxy( coFac );`
  - run two implementations, select result from fastest
  - Groebner base with rational function coefficients, e.g.
    - 3610 subresultant PRS, 2189 modular algorithm was fastest

Parallelization (2)

- Groebner base with work queue of polynomials `CriticalPairList`
  - with synchronized methods `get(), put(), removeNext()` to modify data structure
  - scales well for 8 CPUs on a well structured problem
- distributed version uses some kind of a distributed list to store polynomials of set (implemented by a DHT)
  - use of object serialization for transport of polynomials over the network

Libraries

- advantage of scientific libraries: accumulate knowledge, improve algorithms and implementations
- others
  - jscl-meditor: computer algebra library with GUI front-end [21]
  - Orbital: mathematical logic, Groebner bases [22]
  - JScience: not limited to computer algebra [23]
  - Apache Commons Math: statistics and other utilities missing in Java [38]

Java environment

- earlier computer algebra systems had to develop parts of computer science
- now we can use sophisticated implementations for many relevant data structures
  - lists, trees, maps, arbitrary precision integers
- profit from Java improvements
  - multi-threading, thread safety and inter-networking
  - (parallel) garbage collection
  - 64bit ready
  - virtual machine improvements
  - performance improvements of new JDKs [36]

Conclusions (1)

- sound object oriented design and implementation of a library for algebraic computations
- type safe trough generic type parameters
- as expressive as categories and domains in Axiom due to Java interfaces
- reduced code size and facilitated code reuse
- dependent types limit type safety, but can't be avoided
- all algebraic semantics can be implemented
  - can use checked and unchecked exceptions

Conclusions (2)

- recursive multivariate polynomials allow greatest common divisor implementation
- employs various design patterns, e.g. creational patterns (factory), facade pattern
- object oriented programming looks strange to mathematicians
- used for a large portion of algebraic algorithms
  - a collection of Groebner base algorithms
  - first OO design and implementation of non-commutative polynomials and Groebner bases

Conclusions (3)

- performance comparable to general purpose CAS, but not to special CAS
- working horses are from the Java multi-precision integers and from the collection framework
- Java platform: 64-bit, multi-threading, parallel garbage collection, inter-networking
- Java improvements leverage the performance and capabilities of JAS
- Future
  - more `multiplicative ideal theory', e.g. factorization

ASCM

Thank you

- Questions or Comments?
- http://krum.rz.uni-mannheim.de/jas
- Thanks to
  - Raphael Jolly
  - Thomas Becker, Samuel Kredel
  - Markus Aleksy, Hans-Günther Kruse
  - the referees
  - and other colleagues

ASCM