Generic, Type-safe and Object Oriented Computer Algebra Software

Heinz Kredel, University of Mannheim
Raphael Jolly, Databeans

CASC 2010, Tsakhkadzor

Overview

- Motivation and design considerations
  - Run-Time Systems
  - Object Oriented Software
- Examples
  - Ring Elements and Polynomials
  - Unique Factorization Domains
- Future work and conclusion

Motivation

- software architectural problems with existing CAS
  - monolithic, non modular structure
  - only CLI interfaces to the algorithms
  - ad-hoc run-time memory management
  - non standard interactive scripting languages
- rewrite CAS in object oriented programming and scripting languages
  - Java and Scala vs. Axiom / Aldor
  - are these platforms really suitable ?
  - want to run on new devices and infrastructures

    e.g. Smart phones, Cloud computing

Design considerations

Goal : build on other software projects as much as possible - only the parts specific to computer algebra are to be implemented

Three major parts for computer algebra software:

- **run-time** infrastructure with memory management
- statically typed object oriented algorithm **libraries**
- dynamic interactive **scripting** interpreters (not in this talk)

Run-Time Systems

- constant maintenance and improvements
- more opportunities for code optimization with just-in-time compilers
- memory management with automatic garbage collection
- exception and security constraint handling
- independence of computer hardware and optimization requirements
- suitable for multi-CPU and distributed computing

Object Oriented Software

- usage of contemporary (object oriented) software engineering principles
- modular software architecture, consisting of
  - usage of existing implementations of basic data structures like integers or lists
  - generic type safe algebraic and symbolic algorithm libraries
  - thread safe and multi-threaded library implementations
  - algebraic objects transportable over the network

Object Oriented Software (cont.)

- high performance implementations of algorithms with state of the art asymptotic complexity but also fast and efficient for small problem sizes
- minimizing the ‘abstraction penalty’ which occurs for high-level programming languages compared to low-level assembly-like programming languages

Overview

Motivation and design considerations

Run-Time Systems

Object Oriented Software

Examples

Ring Elements and Polynomials

Unique Factorization Domains

Future work and conclusion

Example : Ring Elements and Polynomials

- polynomials = basis of many algebraic algorithms => are of utmost importance
- devise a 'most' general polynomial class
  - with arbitrary coefficients from some ring
  - which are self usable as coefficients
- polynomials with different types of coefficients should have a different type
- provide abstractions / parametrizations for exponents, memory allocation and more

Element

```
object Element {
    trait Factory[T <: Element[T]] {
        def random(numbits: Int)(implicit rnd:
                                            scala.util.Random): T
    }
}
trait Element[T <: Element[T]] extends Ordered[T] { this:
T =>
    val factory: Element.Factory[T]
    def equals(that: T) = this.compare(that) == 0
    def ><(that: T) = this equals that
    def <>(that: T) = !(this equals that)
}
```

Abelian Group

```
object AbelianGroup {
    trait Factory[T <: AbelianGroup[T]] extends Element.Factory[T] {
        def zero: T
    }
}
trait AbelianGroup[T <: AbelianGroup[T]] extends Element[T] { this: T =>
    override val factory: AbelianGroup.Factory[T]
    def isZero = this >< factory.zero
    def +(that: T): T
    def -(that: T): T
    def unary_+ = this
    def unary_- = factory.zero - this
    def abs = if (signum < 0) -this else this
    def signum: Int
}
```

SemiGroup

```
trait SemiGroup[T <: SemiGroup[T]] extends Element[T] {
this: T =>
    def *(that: T): T
}
```

Monoid

```
object Monoid {
    trait Factory[T <: Monoid[T]] extends Element.Factory[T] {
        def one: T
    }
}
trait Monoid[T <: Monoid[T]] extends SemiGroup[T] { this: T =>
    override val factory: Monoid.Factory[T]
    def isUnit: Boolean
    def isOne = this >< factory.one
    def pow(exp: BigInt) = {
        assert (exp >= 0)
        (factory.one /: (1 to exp.intValue)) {
            (l, r) => l * this
        }
    }
}
```

Ring

```
object Ring {
    trait Factory[T <: Ring[T]] extends
AbelianGroup.Factory[T] with Monoid.Factory[T] {
        def characteristic: BigInt
    }
}
trait Ring[T <: Ring[T]] extends AbelianGroup[T] with
Monoid[T] { this: T =>
    override val factory: Ring.Factory[T]
}
```

Polynomial

```
object Polynomial {
    class Factory[C <: Ring[C]](val ring: C, val variables: Array[Variable], val ordering: Comparator[Int]) extends Ring.Factory[Polynomial[C]] {
        def generators: Array[Polynomial[C]]
        def apply(value: SortedMap[Array[Int], C]) = new Polynomial(this)(value)
        override def toString: String
    }
}
class Polynomial[C <: Ring[C]](val factory: Polynomial.Factory[C])(val value: SortedMap[Array[Int], C]) extends Ring[Polynomial[C]] {
    def elements: Iterator[Pair[Array[Int], C]]
    def headTerm = elements.next
    def degree: Int
    def isUnit = !this.isZero && degree == 0 && headTerm._2.isUnit
    override def toString: String
}
```

Polynomial (cont.)

```
object Polynomial {
    trait Factory[T <: Polynomial[T, C], C <: Ring[C]] extends
Ring.Factory[T] {
        def multiply(w: T, x: Array[Int], y: C) = {
            // commutative case
        }
    }
}
trait Polynomial[T <: Polynomial[T, C], C <: Ring[C]] extends Ring[T]

object SolvablePolynomial {
    trait Factory[T <: Polynomial[T, C], C <: Ring[C]] extends
Polynomial.Factory[T, C] {
        override def multiply(w: T, x: Array[Int], y: C) = {
            // non-commutative case
        }
    }
}
```

Modularity of the design

Elements can be parametrized over:

- the coefficient type (C, above)
- the underlying data structure (array, list, tree)
- the type of the exponents
- the choice of algorithm for gcd computation
- Polynomial or SolvablePolynomial

Not yet implemented:

- improved parametrization of the exponents' type through Scala type specialization
- the list of variables and the ordering (requires dependent types)

Example : Unique Factorization Domains

- exemplify the usefulness of object oriented software for larger algebraic libraries
- algorithms in UFDs to factor polynomials
  - Greatest Common Divisor computation
  - Squarefree decomposition/factorization
  - Factorization
- example: generic factorization over $\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})$

Unique factorization domains

elements of a UFD can be written as $a = u \; p_1^{e_i} \; \ldots \; p_n^{e_n}$ compute this

polynomial rings over UFDs are UFDs $R = UFD[x_1, \; \ldots, \; x_n]$

Gauss Lemma $cont(a\,b) = cont(a) \; cont(b)$

primitive part $a = cont(a) \; pp(a)$

squarefree $\dfrac{a}{gcd(a, a')}$ *is squarefree*

squarefree factorization $a = a_1^1 \; \ldots \; a_d^d$

Greatest Common Divisors

Interface `GreatestCommonDivisor`

abstract class `GreatestCommonDivisorAbstract`

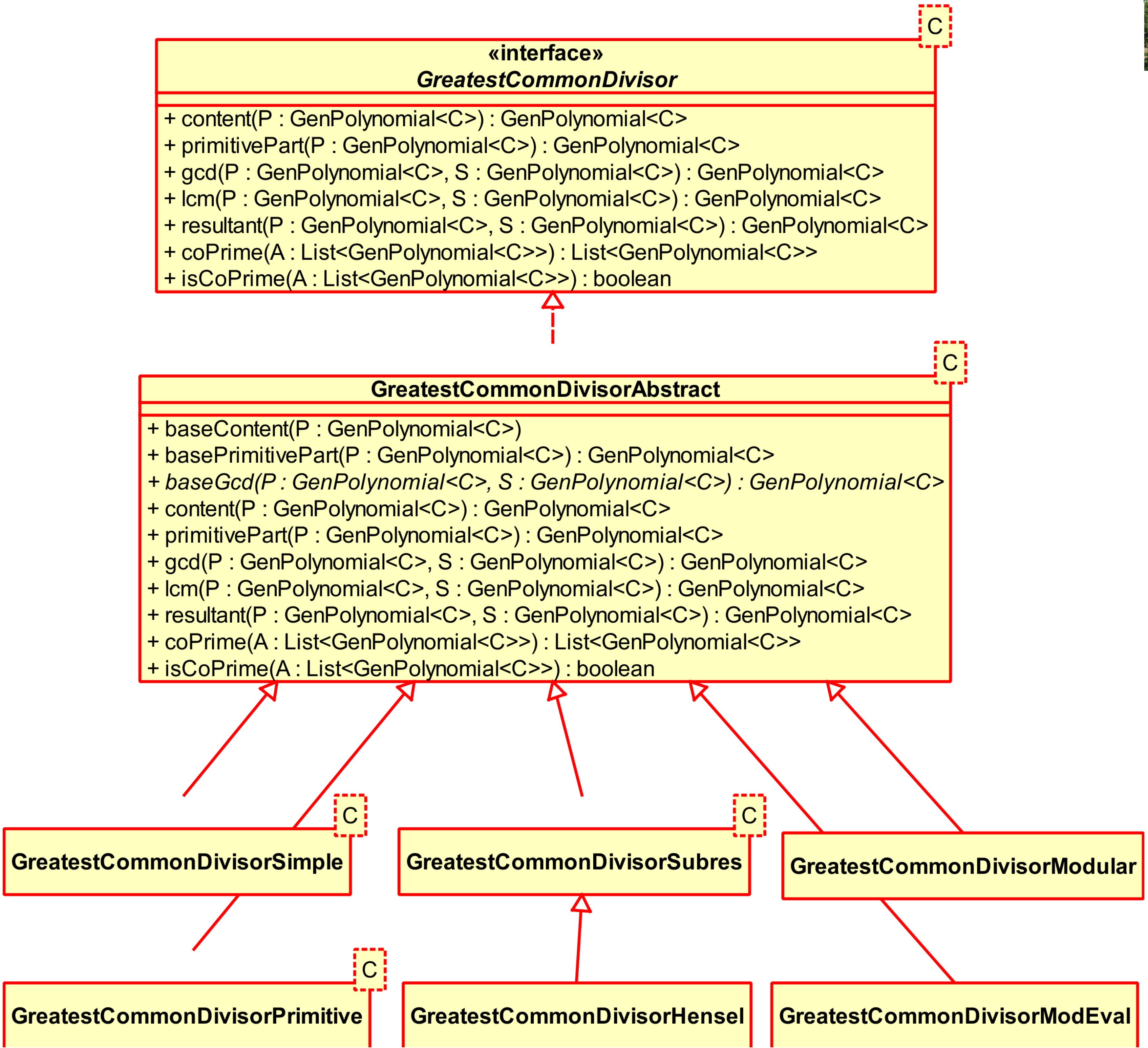
- implements gcd, lcm, content, primitive part, co-prime lists and tests
- `baseGCD()` and `recursiveUnivariateGCD()` are abstract

other classes for different coefficient rings

- generic variants for any field coefficient ring
- modular variants for specific coefficients

GCD implementations

- Polynomial remainder sequences (PRS)
  - primitive PRS
  - simple / monic PRS
  - sub-resultant PRS
- modular methods
  - modular coefficients, Chinese remaindering (CR)
  - recursion by modular evaluation and CR
  - modular coefficients, Hensel lifting wrt. p^e
  - recursion by multivariate Hensel lifting


C
«interface»
GreatestCommonDivisor
+ content(P : GenPolynomial<C>) : GenPolynomial<C>
+ primitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ gcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ lcm(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ resultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ coPrime(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ isCoPrime(A : List<GenPolynomial<C>>) : boolean
C
GreatestCommonDivisorAbstract
+ baseContent(P : GenPolynomial<C>)
+ basePrimitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ baseGcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ content(P : GenPolynomial<C>) : GenPolynomial<C>
+ primitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ gcd(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ lcm(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ resultant(P : GenPolynomial<C>, S : GenPolynomial<C>) : GenPolynomial<C>
+ coPrime(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ isCoPrime(A : List<GenPolynomial<C>>) : boolean
C
GreatestCommonDivisorSimple
C
GreatestCommonDivisorSubres
GreatestCommonDivisorModular
C
GreatestCommonDivisorPrimitive
GreatestCommonDivisorHensel
GreatestCommonDivisorModEval

GCD factory

- all gcd variants have pros and cons
  - computing times differ in a wide range
  - coefficient rings enable specific treatment
- solve by object-oriented factory design pattern: a factory class creates and provides a suitable implementation via different methods

```
GreatestCommonDivisor<C>
  GCDFactory.<C>getImplementation( cfac );
```

  - type C and type of cfac triggers selection at compile time
  - coefficient factory cfac triggers selection at runtime

GCD proxy

- variable performance of algorithms
  - mostly modular methods are faster
  - but some times (sub-resultant) PRS faster
- hard to predict run-time of algorithm for inputs
- improvement by speculative parallelism
- execute two (or more) algorithms in parallel

`java.util.concurrent.ExecutorService.invokeAny()`

  - executes several methods in parallel
  - when one finishes the others are interrupted

Squarefree decomposition

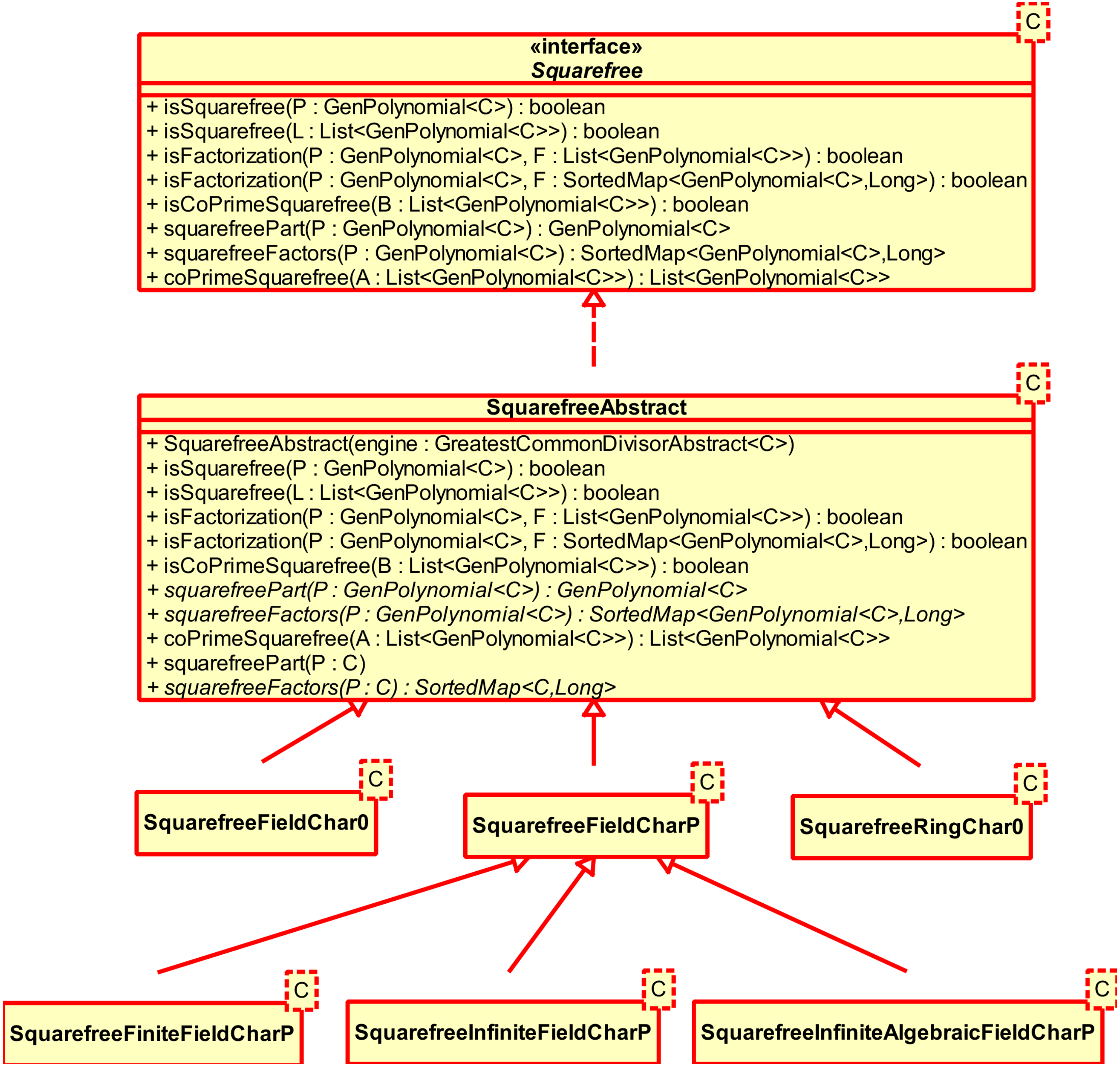
interface `Squarefree`

abstract class `SquarefreeAbstract`

- implements tests and co-prime squarefree set construction
- `squarefreeFactors()`, `squarefreePart()` abstract

other classes for different coefficient rings

- – ring or fields of characteristic zero
- – fields of characteristic p > 0
  - finite fields
  - infinite fields, transcendental extensions
  - algebraic extensions of infinite fields


C
«interface»
Squarefree
+ isSquarefree(P : GenPolynomial<C>) : boolean
+ isSquarefree(L : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : SortedMap<GenPolynomial<C>,Long>) : boolean
+ isCoPrimeSquarefree(B : List<GenPolynomial<C>>) : boolean
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ coPrimeSquarefree(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
C
SquarefreeAbstract
+ SquarefreeAbstract(engine : GreatestCommonDivisorAbstract<C>)
+ isSquarefree(P : GenPolynomial<C>) : boolean
+ isSquarefree(L : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : SortedMap<GenPolynomial<C>,Long>) : boolean
+ isCoPrimeSquarefree(B : List<GenPolynomial<C>>) : boolean
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ coPrimeSquarefree(A : List<GenPolynomial<C>>) : List<GenPolynomial<C>>
+ squarefreePart(P : C)
+ squarefreeFactors(P : C) : SortedMap<C,Long>
C
SquarefreeFieldChar0
C
SquarefreeFieldCharP
C
SquarefreeRingChar0
C
SquarefreeFiniteFieldCharP
C
SquarefreeInfiniteFieldCharP
C
SquarefreeInfiniteAlgebraicFieldCharP

Squarefree factory

- selection based on given type parameter and coefficient ring factory
- generic relative to characteristic of the ring
- special cases for characteristic p > 0
  - transcendental field extensions, coefficients from class `Quotient` `SquarefreeInfiniteFieldCharP`
  - algebraic field extensions of transcendental extensions, coefficients from class `AlgebraicNumber` `SquarefreeInfiniteAlgebraicField-CharP`

Factorization

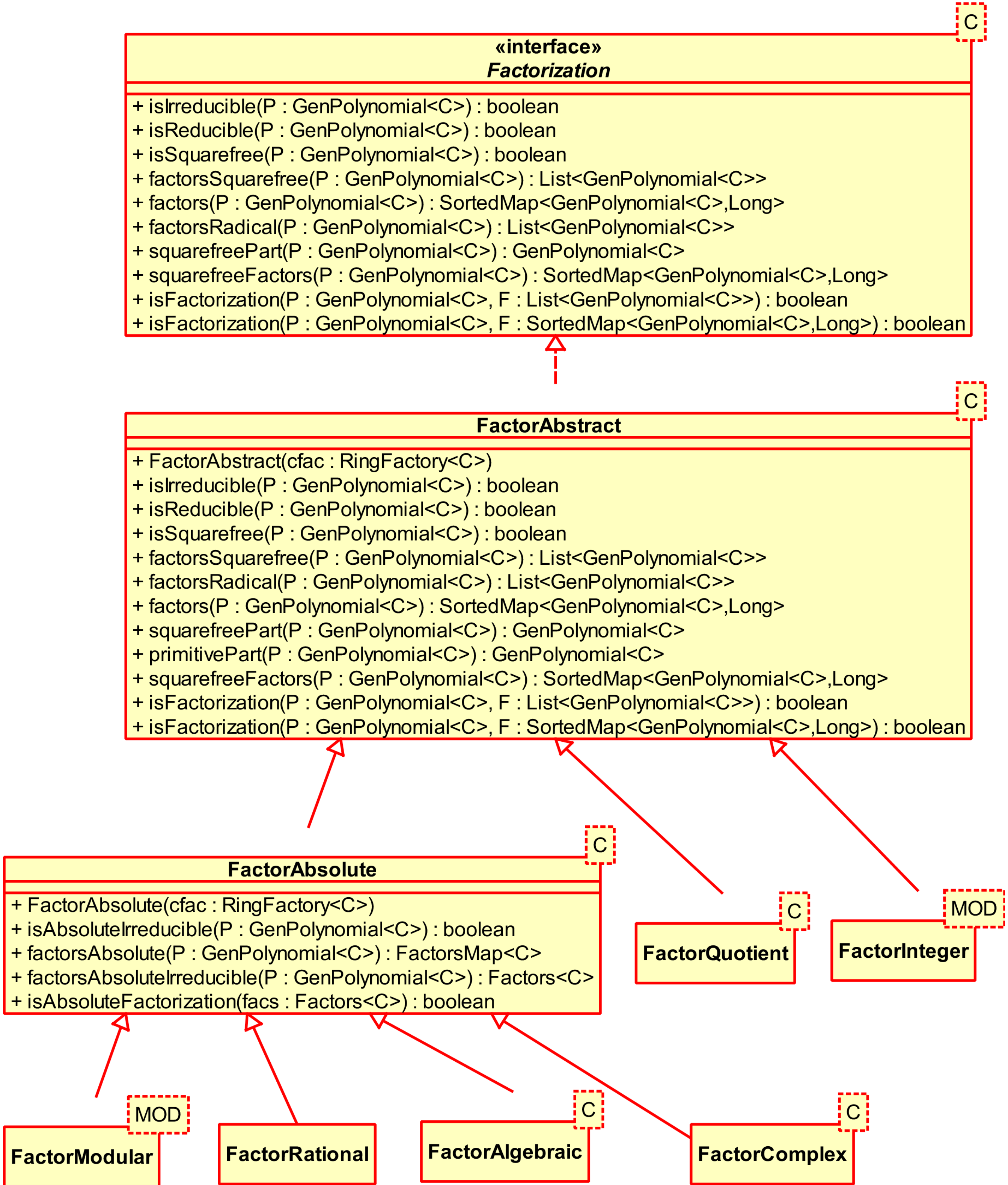
interface `Factorization`

abstract class `FactorAbstract`

- implements nearly everything, only `baseFactorSquarefree()` must be implemented for each coefficient ring
- uses (slow) Kronecker substitution for reduction to univariate case and multivariate reconstruction
  - multivariate Hensel lifting in the future

class `FactorAbsolute` for splitting fields

- extend coefficient ring until factors become linear
- abstract and intermediate between `FactorAbstract`



C
«interface»
Factorization
+ isIrreducible(P : GenPolynomial<C>) : boolean
+ isReducible(P : GenPolynomial<C>) : boolean
+ isSquarefree(P : GenPolynomial<C>) : boolean
+ factorsSquarefree(P : GenPolynomial<C>) : List<GenPolynomial<C>>
+ factors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ factorsRadical(P : GenPolynomial<C>) : List<GenPolynomial<C>>
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ isFactorization(P : GenPolynomial<C>, F : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : SortedMap<GenPolynomial<C>,Long>) : boolean
C
FactorAbstract
+ FactorAbstract(cfac : RingFactory<C>)
+ isIrreducible(P : GenPolynomial<C>) : boolean
+ isReducible(P : GenPolynomial<C>) : boolean
+ isSquarefree(P : GenPolynomial<C>) : boolean
+ factorsSquarefree(P : GenPolynomial<C>) : List<GenPolynomial<C>>
+ factorsRadical(P : GenPolynomial<C>) : List<GenPolynomial<C>>
+ factors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ squarefreePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ primitivePart(P : GenPolynomial<C>) : GenPolynomial<C>
+ squarefreeFactors(P : GenPolynomial<C>) : SortedMap<GenPolynomial<C>,Long>
+ isFactorization(P : GenPolynomial<C>, F : List<GenPolynomial<C>>) : boolean
+ isFactorization(P : GenPolynomial<C>, F : SortedMap<GenPolynomial<C>,Long>) : boolean
C
FactorAbsolute
+ FactorAbsolute(cfac : RingFactory<C>)
+ isAbsoluteIrreducible(P : GenPolynomial<C>) : boolean
+ factorsAbsolute(P : GenPolynomial<C>) : FactorsMap<C>
+ factorsAbsoluteIrreducible(P : GenPolynomial<C>) : Factors<C>
+ isAbsoluteFactorization(facs : Factors<C>) : boolean
C
FactorQuotient
MOD
FactorInteger
MOD
FactorModular
FactorRational
C
FactorAlgebraic
C
FactorComplex

Factorization (cont.)

`FactorModular`

- implements `distinctDegreeFactor()` and `equalDegreeFactor()`

  Berlekamp algorithm in the future

`FactorInteger`

- computes modulo primes, lifts with Hensel and does combinatorial factor search

`FactorRational`

- clears denominators and uses factorization over integers

Factorization (cont.)

`FactorAlgebraic`

- for algebraic field extensions for arbitrary coefficient rings
  - first for modular and rational coefficients
  - also for `Quotient` and `AlgebraicNumber`
- computes norm, then factors norm
- use gcds between factors of norm and polynomial

`FactorQuotient`

- for transcendental field extensions for arbitrary coefficient rings
- clears denominators, then factors multivariate polynomial over the next coefficient ring

Factorization factory

- selection based on given type parameter and coefficient ring factory
- implementations for mentioned coefficient rings
- generic cases for polynomial coefficients
  - transcendental field extensions, coefficients from class `Quotient`: `FactorQuotient`
  - algebraic field extensions, coefficients from class `AlgebraicNumber`: `FactorAlgebraic`
- factory used to select implementation step by step as coefficient rings uncover

  see following example

Factorization example

mathematical Ring

$$\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})[y]$$

Ring in jython

```
PolyRing(AN(( wx**2 - x ),True,
    PolyRing(RF(
        PolyRing(AN(( w2**2 -2 ),True,
            PolyRing(QQ(),"w2",PolyRing.lex)),
        "x",PolyRing.lex)),
    "wx",PolyRing.lex)),
"y",PolyRing.lex)
```

polynomial

f = y**4 - (x + 2) y**2 + 2 x = (y**2 – x) (y**2 - 2)

factors

```
h   =  ( y - wx )
h   =  ( y - w2 )
h   =  ( y + wx )
h   =  ( y + w2 )
```

$$wx = \sqrt{x}$$

$$w2 = \sqrt{2}$$

factor time = 11168 milliseconds

Example is in: examples/factors_algeb_trans.py

Categorical factorization

- first in Scratchpad (now Axiom)
- factorization of multivariate polynomials over arbitrarily nested coefficient rings provided there is an algorithm for univariate polynomials over such a coefficient ring
- sequence of factorizations
- selection via factory

$\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})[y]$ given

$\to (\mathbb{Q}(\sqrt{2})(x))(\sqrt{x})[y]$ algebraic

$\to \mathbb{Q}(\sqrt{2})(x)[wx]$ transcendent

$\to \mathbb{Q}(\sqrt{2})[x, wx] \to \mathbb{Q}(\sqrt{2})[z]$

$\to \mathbb{Q}[w2, z] \to \mathbb{Q}[z'] \to \mathbb{Z}[z'] \to \mathbb{Z}_p[z']$

Overview

Motivation and design considerations

Run-Time Systems

Object Oriented Software

Examples

Ring Elements and Polynomials

Unique Factorization Domains

Future work and conclusion

Future work

(remaining) problems with existing object oriented languages

- interface type parameter adaption in sub-classes
- “extend” polynomial type by
  - number of variables
  - names of variables
- enhance polynomial implementation by gcd or factorization vs. separate class hierarchies for gcd or factorization

Future work (cont.)

- using existing rich client platforms like Eclipse by MathEclipse
- using webservice and Cloud computing platforms, like Google App Engine by Symja
- use the scripting ability of Android to make computer algebra available on mobile phones
- define a common interface with Apache Commons Math and/or JLinAlg
- Maxima, Reduce could run on the JVM
- MathML/OpenMath easy to use in Java

Conclusions (1)

- can concentrate on mathematical aspects by
  - re-using software components
  - Java and Scala language with JVM run-time
  - interactive scripting languages
- JVM infrastructure opens new ways of
  - interoperability of computer algebra systems on Java byte-code level
  - gives also new opportunities to provide CAS
    - on new computing devices
    - software as a service
    - distributed or cloud computing

Conclusions (2)

- CAS design and implementation by leveraging 30 years of advances in computer science
- object oriented approach with the Java and Scala programming languages
  - can implement non-trivial algebraic structures
  - in a type-safe way
  - with competitive performance
  - can be stacked and plugged together in various ways

Thank you for your attention

Questions ?

Comments ?

http://jscl-meditor.sourceforge.net/

http://krum.rz.uni-mannheim.de/jas/

Acknowledgments

- thanks to: Thomas Becker, Werner K. Seiler, Axel Kramer, Dongming Wang, Thomas Sturm, Hans-Günther Kruse, Markus Aleksy
- thanks to the referees