Generic and parallel Gröbner bases in JAS

Heinz Kredel, University of Mannheim

4th International Congress on Mathematical Software

August 2014, Hanyang University, Seoul, Korea

Overview

- Introductory example
- Generic Gröbner bases
    - interface and abstract class
    - sequential algorithm
    - parallel and distributed algorithms
- Implementation selection and composition
    - selection for a coefficient ring
    - composition of implementations
- Conclusions

Introductory example

- polynomial ring over a field tower

$$R = E[y, z] = \mathbb{Q}(\sqrt{2})(x)(\sqrt{x})[y, z],$$

- corresponding coefficient type in Java

```
AlgebraicNumber<Quotient<AlgebraicNumber<BigRational>>>
```

- the construction of elements is provided via factories called `...Ring`

```
AlgebraicNumberRing<Quotient<
               AlgebraicNumber<BigRational>>> cfac = ...
```

Example (compute GB)

- obtain Gröbner base implementation for this coefficient ring, setup polynomial lists and compute Gröbner base

```
GroebnerBase<
     AlgebraicNumber<Quotient<AlgebraicNumber<BigRational>>>> bb;
bb = GBFactory.getImplementation(cfac);

List<
 GenPolynomial<AlgebraicNumber<Quotient<AlgebraicNumber<BigRational>>>>
> G, F = ...;

G = bb.GB(F);

System.out.println("isGB(G) = " + bb.isGB(G));
```

Example (simplified)

- algebraic constructions can be done also within Gröbner base computation

$\mathbb{Q}(x)[w_2, w_x, y\underset{\longleftarrow}{,\ } z]$ add $w_2^2 - 2$ $w_x^2 - x$

```
Quotient<BigRational>
QuotientRing<BigRational> qfac = …;

GroebnerBaseAbstract<Quotient<BigRational>> bb;
bb = GBFactory.getImplementation(qfac);

List<GenPolynomial<Quotient<BigRational>>> G, F = …;
// add w2^2 - 2 and wx^2 - x to F

G = bb.GB(F);
```

Java Algebra System (JAS)

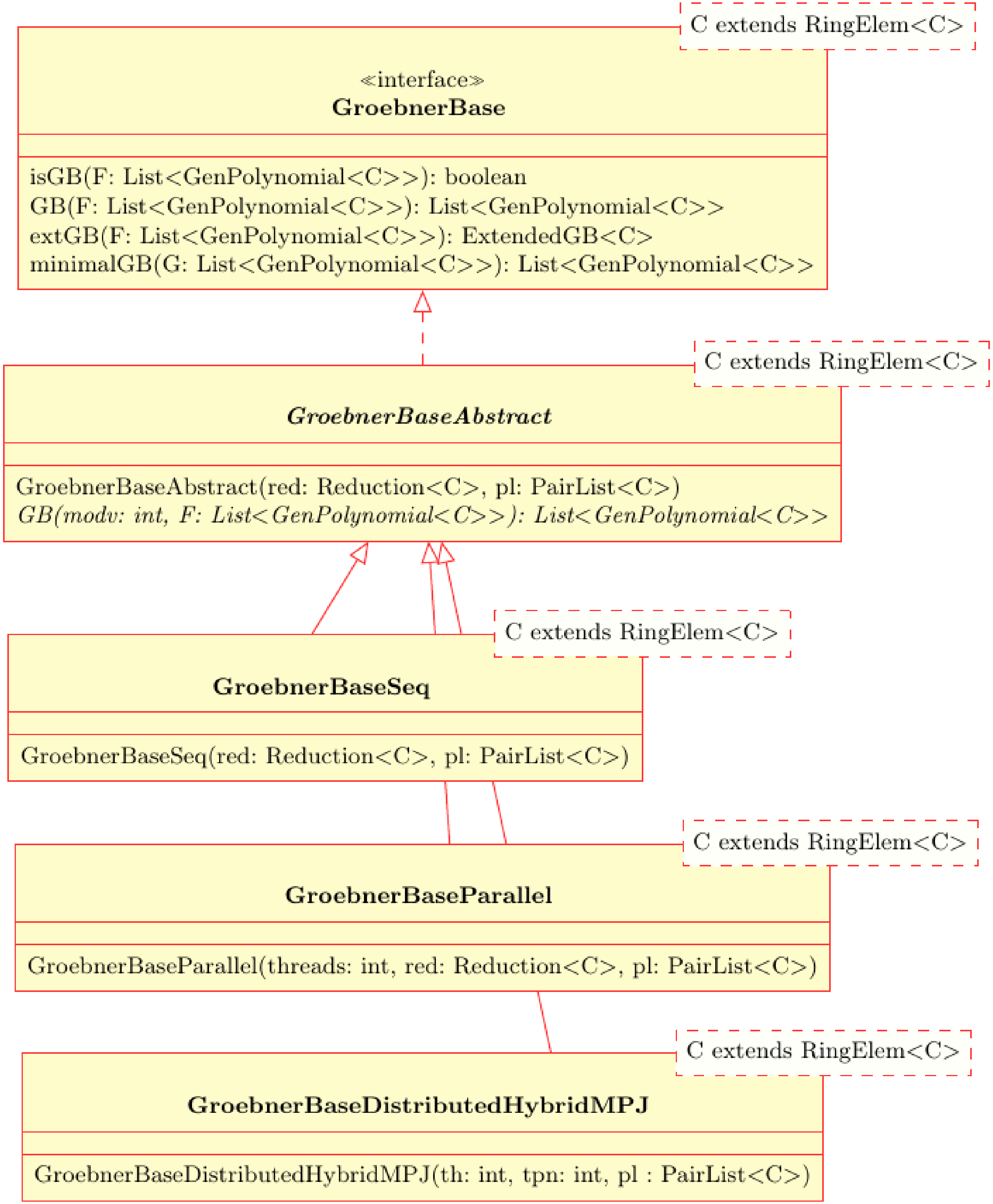
- generic multivariate polynomial rings
- generic implementations of various algorithms
  - Gröbner bases, greatest common divisors
  - factorization, non-commutative rings
- object oriented design of a computer algebra system
  - type safe through Java generic types
- leverage software and hardware improvements
  - multi-threading, parallel Garbage collection
  - multi-core CPUs, compute clusters

Overview

- Introductory example
- Generic Gröbner bases
    - interface and abstract class
    - sequential algorithm
    - parallel and distributed algorithms
- Implementation selection and composition
    - selection for a coefficient ring
    - composition of implementations
- Conclusions

Generic Gröbner bases

- depending on coefficient rings of polynomial rings
    - fields
    - rings with pseudo division
    - regular rings
- sequential, parallel and distributed computing environments
- cases using transformations
    - change of coefficient ring
    - change of term order
- new algorithms, e.g. signature based GBs



C extends RingElem<C>

«interface»
GroebnerBase

isGB(F: List<GenPolynomial<C>>): boolean
GB(F: List<GenPolynomial<C>>): List<GenPolynomial<C>>
extGB(F: List<GenPolynomial<C>>): ExtendedGB<C>
minimalGB(G: List<GenPolynomial<C>>): List<GenPolynomial<C>>

C extends RingElem<C>

GroebnerBaseAbstract

GroebnerBaseAbstract(red: Reduction<C>, pl: PairList<C>)
GB(modv: int, F: List<GenPolynomial<C>>): List<GenPolynomial<C>>

C extends RingElem<C>

GroebnerBaseSeq

GroebnerBaseSeq(red: Reduction<C>, pl: PairList<C>)

C extends RingElem<C>

GroebnerBaseParallel

GroebnerBaseParallel(threads: int, red: Reduction<C>, pl: PairList<C>)

C extends RingElem<C>

GroebnerBaseDistributedHybridMPJ

GroebnerBaseDistributedHybridMPJ(th: int, tpn: int, pl : PairList<C>)

GroebnerBase interface

- generic type parameter `C`:
  - `C` extends `RingElem<C>`
    - includes a `inverse()` method
  - `RingFactory` provides `isField()`
- method parameters: `List<GenPolynomials<C>>`
- test for Gröbner base: `isGB(.)`
- compute a Gröbner base: `GB(.)`
- compute a Gröbner base together with back and forward transformations: `extGB(.)`
- compute a minimal reduced Gröbner base from a Gröbner base: `minimalGB(.)`

GroebnerBaseAbstract

- implements all methods from interface
- abstract method: `GB(modv: int; F: List<.>)`
    - modv: number of module variables, for the computation of module Gröbner bases
- constructor injects implementations for desired polynomial reduction and book-keeping for pair-list
    - `Reduction` parameter
        - methods `normalform(.,.)` and `SPolynomial(.,.)`
    - `PairList` parameter
        - `put(poly)`
        - `removeNext(): Pair`
        - `hasNext(): boolean`

GroebnerBaseSeq

- implements `GB(modv: int, F: List<.>)`
- inherits other methods
- critical pair list implemented as thread-safe working queues (in shared memory for parallel and distributed versions)
- implementations of `PairList` for different selection strategies
  - `OrderedPairlist`, optimized Buchberger
  - `OrderedSyzPairlist`, Gebauer-Möller version
  - `CriticalPairlist`, stay similar to sequential

GroebnerBaseParallel

- implements `GB(modv: int, F: List<.>)`
- uses Java threads for expensive `normalform()`
    - number of threads via constructor parameter
- polynomial list is kept in shared memory and concurrently used by all threads
- `ReductionPar` implements `Reduction`, tolerates asynchronous updates of polynomial list
- correct termination detection subtle
- new polynomials appear in different sequence order than in sequential algorithm

GroebnerBaseDistributedHybrid

- implements `GB(modv: int, F: List<.>)`
- inherits other methods
- uses distributed memory computers with multi-core compute nodes
- supported environments
  - Java TCP/IP Sockets also with newio
  - MPJ (FastMPJ, MPJ Express)
    - pure Java and direct InfiniBand interconnect
  - OpenMPI with Java bindings
- PBS job handling system

GroebnerBaseDistributedHybrid

- list of reduction polynomials
  - replicated to all compute nodes
  - in shared memory on each node
- threads on compute nodes
  - receive critical pairs from master node
  - send reduction polynomials to master
- pair list maintained on master node
- termination detection on master node
- polynomial transport using Java object serialization

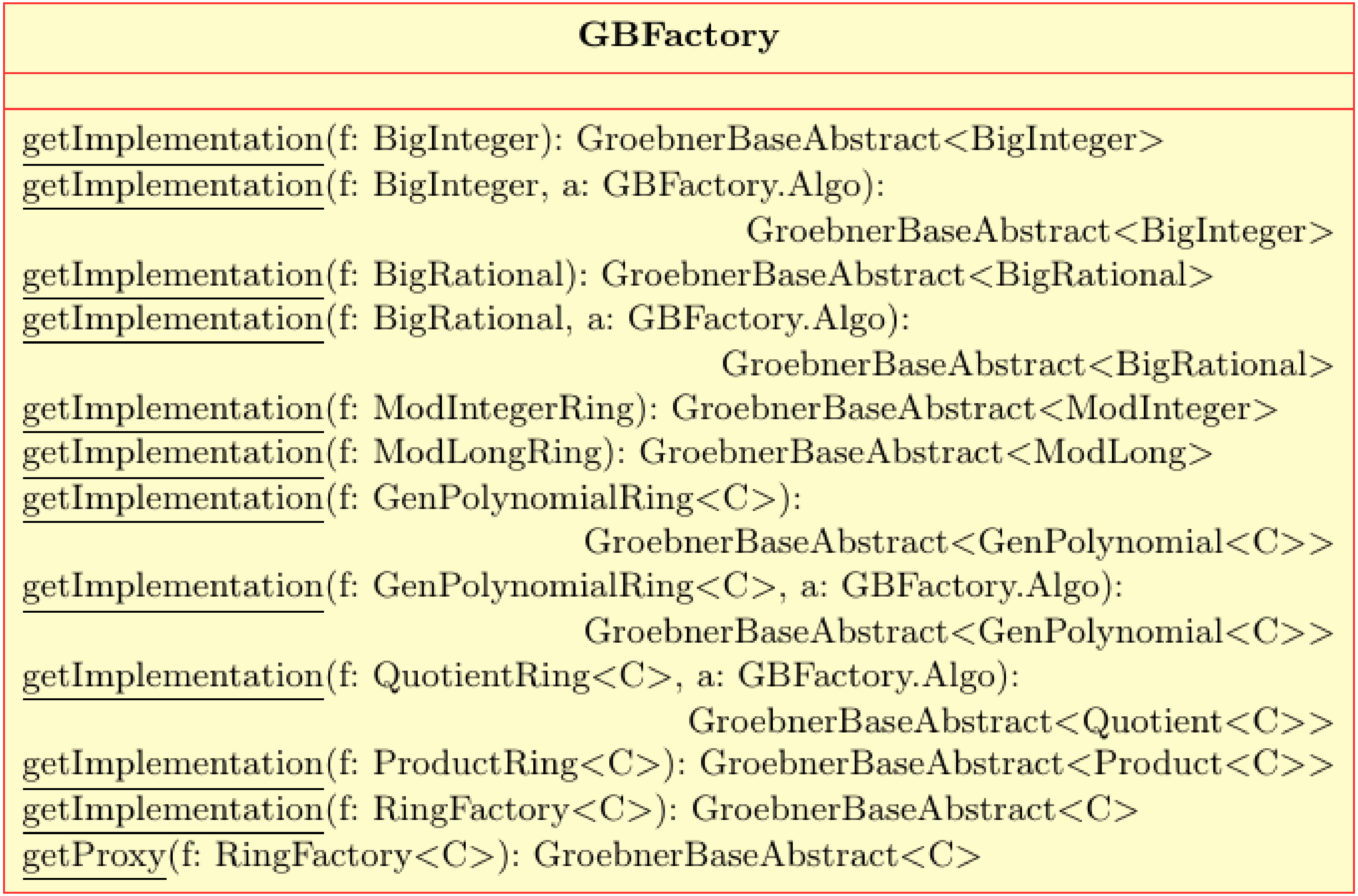
Overview

- Introductory example
- Generic Gröbner bases
    - interface and abstract class
    - sequential algorithm
    - parallel and distributed algorithms
- Implementation selection and composition
    - selection for a coefficient ring
    - composition of implementations
- Conclusions

Selection of an implementation

- `GBFactory`: a way to select an implementaton of an algorithm for Gröbner base computation
- provides static polymorphic methods `getImplementation(.)`
- for different coefficient rings
  - `BigInteger, BigRational, ModInteger, ModLong,`
  - `QuotientRing<C>, ProductRing<C>`
  - generic `RingFactory<C>`
- returns object of type `GroebnerBaseAbstract<C>`
- `getProxy(.)` provides parallel implementation

Gröbner base factory



GBFactory

getImplementation(f: BigInteger): GroebnerBaseAbstract<BigInteger>

getImplementation(f: BigInteger, a: GBFactory.Algo): GroebnerBaseAbstract<BigInteger>

getImplementation(f: BigRational): GroebnerBaseAbstract<BigRational>

getImplementation(f: BigRational, a: GBFactory.Algo): GroebnerBaseAbstract<BigRational>

getImplementation(f: ModIntegerRing): GroebnerBaseAbstract<ModInteger>

getImplementation(f: ModLongRing): GroebnerBaseAbstract<ModLong>

getImplementation(f: GenPolynomialRing<C>): GroebnerBaseAbstract<GenPolynomial<C>>

getImplementation(f: GenPolynomialRing<C>, a: GBFactory.Algo): GroebnerBaseAbstract<GenPolynomial<C>>

getImplementation(f: QuotientRing<C>, a: GBFactory.Algo): GroebnerBaseAbstract<Quotient<C>>

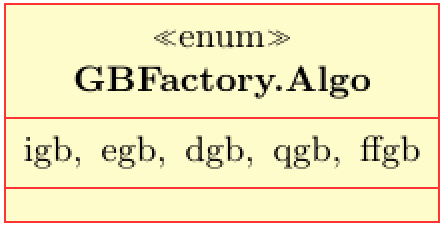
getImplementation(f: ProductRing<C>): GroebnerBaseAbstract<Product<C>>

getImplementation(f: RingFactory<C>): GroebnerBaseAbstract<C>

getProxy(f: RingFactory<C>): GroebnerBaseAbstract<C>

GB Algo

- for `BigRational` and `QuotientRing<C>`
    - fraction/quotient coefficients “`qgb`”
    - fraction free coefficients “`ffgb`”
- for `BigInteger` and univariate `GenPolynomial<C>` over field
    - pseudo division “`igb`”
    - d- or e-Gröbner base “`dgb, egb`”



GBProxy

- `GBProxy` extends GroebnerBaseAbstract
- constructor accepts two `GroebnerBaseAbstract` parameters
- the `GB(modv, .)` method executes both corresponding `GB(modv, .)` methods in parallel
- based on `java.util.concurrent.ExecutorService`
- method `invokeAny(.,.)` returns result of first finished computation and cancels the other one
- with a sequential and parallel Gröbner base
    - for small problems sequential is often faster
    - for larger problems and multi-cores parallel

Example

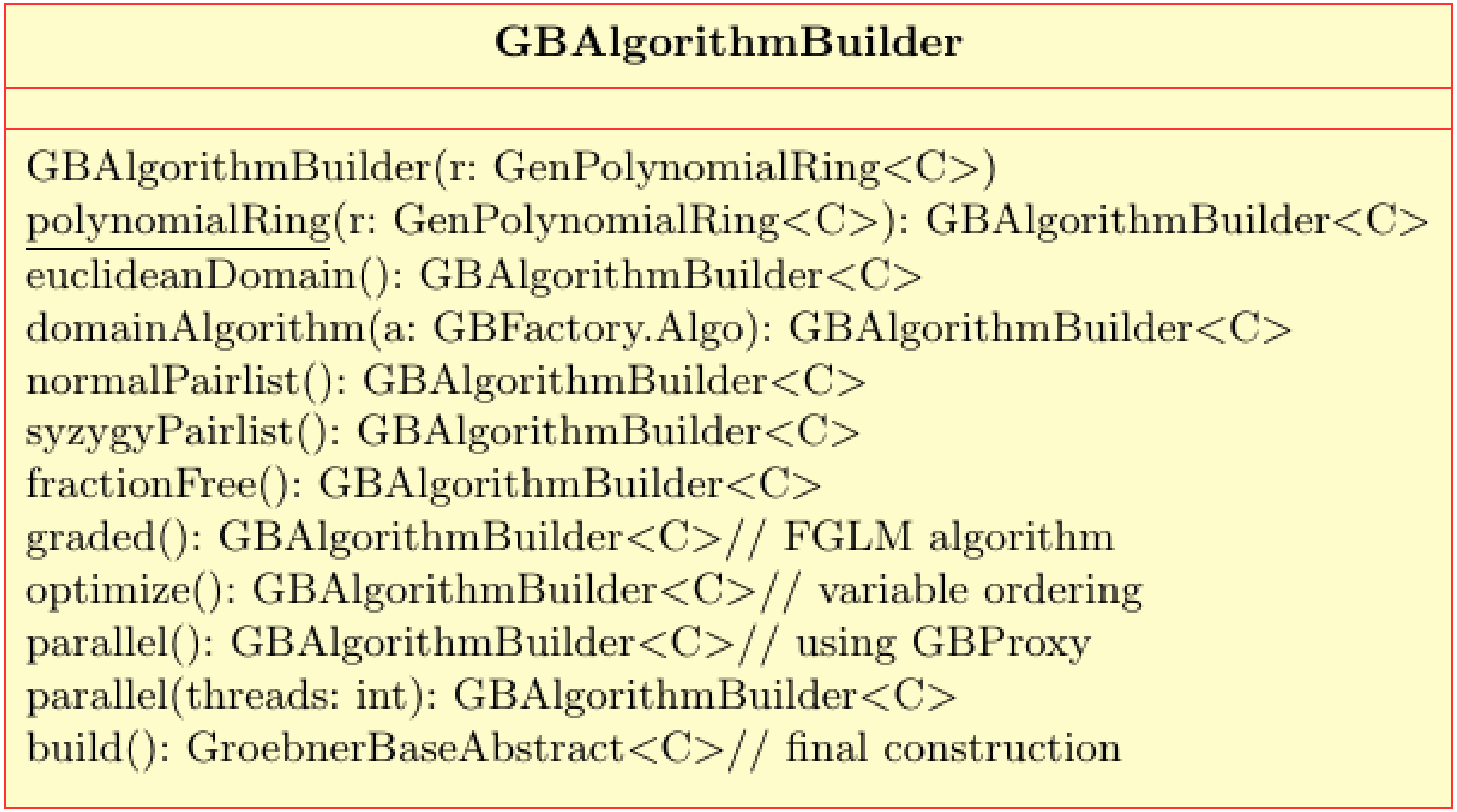
- example of a parallel computation

```
GroebnerBaseAbstract<Quotient<BigRational>> bb;

bb = GBFactory.getProxy(qfac);
     // get a parallel implementation

List<GenPolynomial<Quotient<BigRational>>> G, F
= ...;

G = bb.GB(F);
```

Composition of implementations

- further variants of Gröbner base algorithms
  - transformation of coefficient rings, quotient or fraction free
  - transformation of term order, FGLM algorithm
  - optimize term order
  - select pair list strategy
- such variants can be combined
  - start with definition of first coefficient ring
  - compose variants as desired or possible
  - finalize composition with `build()` method
- implemented in GBAlgorithmBuilder

GB Algorithm Builder



GBAlgorithmBuilder

GBAlgorithmBuilder(r: GenPolynomialRing<C>)
polynomialRing(r: GenPolynomialRing<C>): GBAlgorithmBuilder<C>
euclideanDomain(): GBAlgorithmBuilder<C>
domainAlgorithm(a: GBFactory.Algo): GBAlgorithmBuilder<C>
normalPairlist(): GBAlgorithmBuilder<C>
syzygyPairlist(): GBAlgorithmBuilder<C>
fractionFree(): GBAlgorithmBuilder<C>
graded(): GBAlgorithmBuilder<C>// FGLM algorithm
optimize(): GBAlgorithmBuilder<C>// variable ordering
parallel(): GBAlgorithmBuilder<C>// using GBProxy
parallel(threads: int): GBAlgorithmBuilder<C>
build(): GroebnerBaseAbstract<C>// final construction

Example

- composition in case of FGLM algorithm
  `GroebnerBaseFGLM(GroebnerBaseAbstract .)`
- FGLM: `graded()`
- term order optimization: `optimize()`
- example: compose fraction free and parallel GB

```
GenPolynomialRing<Quotient<BigRational>> pfac = ...

bb = GBAlgorithmBuilder.polynomialRing(pfac)
     .fractionFree().parallel(5).build();

List<GenPolynomial<BigRational>> G, F = ...;
G = bb.GB(F);
```

Conclusions

- JAS: basic software for polynomial rings with generic coefficient rings
- generic implementations of Gröbner base computation and others like factorization
- user friendly selection of suitable implementations with `GBFactory`
- user friendly composition of variants of Gröbner base implementation: parallel, FGLM, optimization, pair list selection
- parallel algorithm on multi-core computers
- distributed algorithm for compute clusters

Thank you for your attention

Questions ?

Comments ?

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

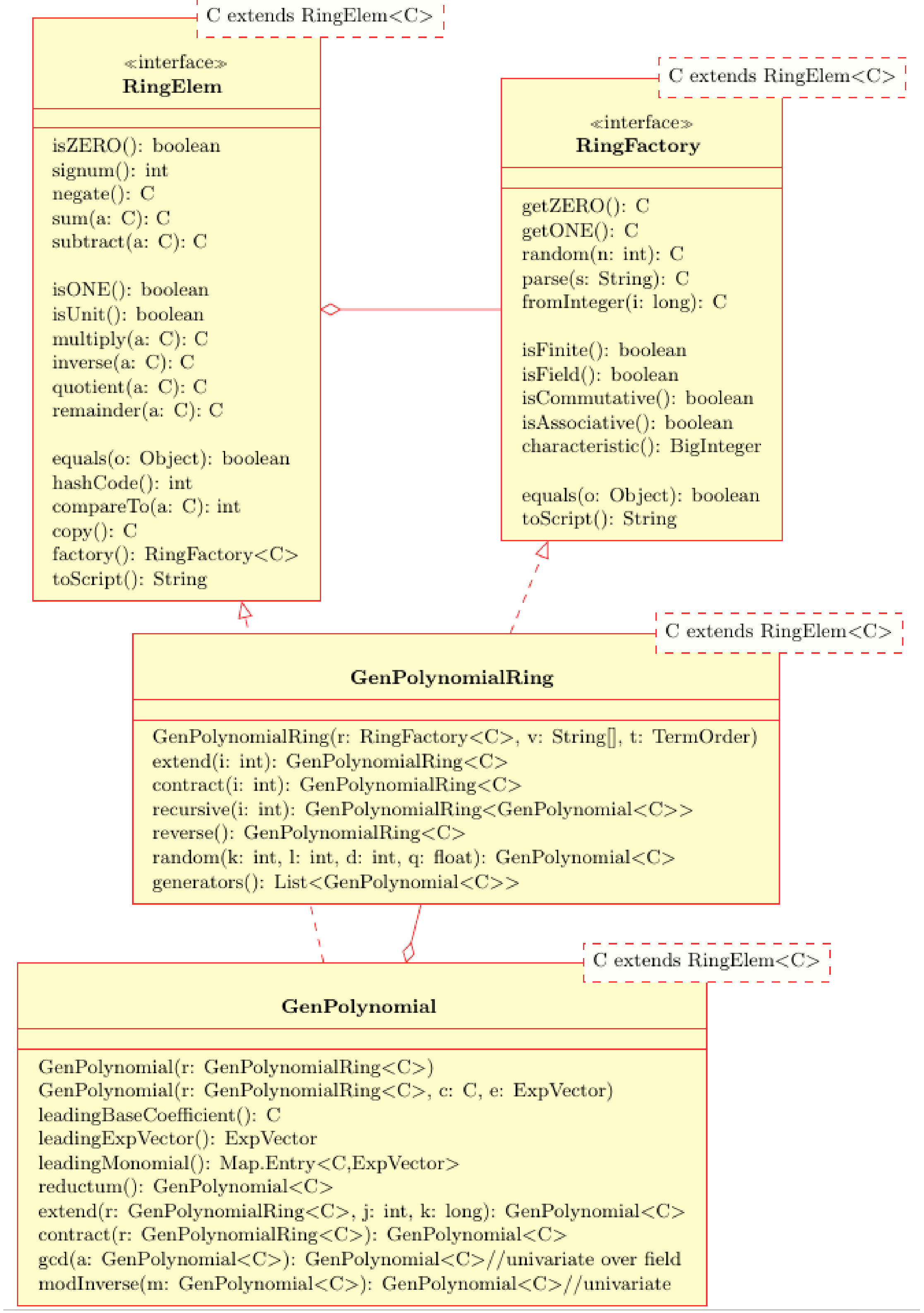
Acknowledgements

- thanks to: Thomas Becker, Raphael Jolly, Wolfgang K. Seiler, Axel Kramer, Dongming Wang, Thomas Sturm, Hans-Günther Kruse, Markus Aleksy

more slides

JAS Implementation overview

- 375+ classes and interfaces
- plus ~170 JUnit test classes,1000+ unit tests
- uses JDK 1.7 with generic types
  - Javadoc API documentation
  - logging with Apache Log4j
  - build tool is Apache Ant
  - revision control with Subversion
  - public git repository
- jython (Java Python), jruby (Java Ruby) scripts
  - support for Sage compatible polynomial expressions
- Android version based on Ruboto using jruby



C extends RingElem<C>

«interface»
RingElem

isZERO(): boolean
signum(): int
negate(): C
sum(a: C): C
subtract(a: C): C

isONE(): boolean
isUnit(): boolean
multiply(a: C): C
inverse(a: C): C
quotient(a: C): C
remainder(a: C): C

equals(o: Object): boolean
hashCode(): int
compareTo(a: C): int
copy(): C
factory(): RingFactory<C>
toScript(): String

C extends RingElem<C>

«interface»
RingFactory

getZERO(): C
getONE(): C
random(n: int): C
parse(s: String): C
fromInteger(i: long): C

isFinite(): boolean
isField(): boolean
isCommutative(): boolean
isAssociative(): boolean
characteristic(): BigInteger

equals(o: Object): boolean
toScript(): String

C extends RingElem<C>

GenPolynomialRing

GenPolynomialRing(r: RingFactory<C>, v: String[], t: TermOrder)
extend(i: int): GenPolynomialRing<C>
contract(i: int): GenPolynomialRing<C>
recursive(i: int): GenPolynomialRing<GenPolynomial<C>>
reverse(): GenPolynomialRing<C>
random(k: int, l: int, d: int, q: float): GenPolynomial<C>
generators(): List<GenPolynomial<C>>

C extends RingElem<C>

GenPolynomial

GenPolynomial(r: GenPolynomialRing<C>)
GenPolynomial(r: GenPolynomialRing<C>, c: C, e: ExpVector)
leadingBaseCoefficient(): C
leadingExpVector(): ExpVector
leadingMonomial(): Map.Entry<C,ExpVector>
reductum(): GenPolynomial<C>
extend(r: GenPolynomialRing<C>, j: int, k: long): GenPolynomial<C>
contract(r: GenPolynomialRing<C>): GenPolynomial<C>
gcd(a: GenPolynomial<C>): GenPolynomial<C>//univariate over field
modInverse(m: GenPolynomial<C>): GenPolynomial<C>//univariate