Algebraic structures as typed objects

Heinz Kredel[1] and Raphael Jolly[2]

[1] IT-Center, University of Mannheim, Germany
[2] Databeans, Paris, France
kredel@rz.uni-mannheim.de, raphael.jolly@free.fr

Abstract. Following the research direction of strongly typed, generic, object oriented computer algebra software, we examine the modeling of algebraic structures as typed objects in this paper. We discuss the design and implementation of algebraic and transcendental extension fields together with the modeling of real algebraic and complex algebraic extension fields. We will show that the modeling of the relation between algebraic and real algebraic extension fields using the delegation design concept has advantages over the modeling as sub-types using sub-class implementation. We further present a summary of design problems, which we have encountered so far with our implementation in Java and present possible solutions in Scala.

1 Introduction

We proposed a software architecture for computer algebra systems which builds on other software projects as much as possible in [1] . Only the parts specific to computer algebra are then to be implemented. We identified three major parts for computer algebra software.

– run-time infrastructure with memory management and parallelism,
– statically typed object oriented algorithm libraries,
– dynamic interactive scripting interpreters.

In this paper we elaborate on the second point: the research area of strongly typed, generic, object oriented computer algebra systems, namely the modeling of algebraic structures as typed objects. A prominent part of algebra deals with the study of field extensions. To make use of extension fields in computers we need to restrict ourselves to effective constructions which can be performed on a computer. Such fields are named *computable fields.*

Given some base fields, especially the prime fields, like rational numbers or modular integers (computing modulo a prime number), we will examine the design and implementation of algebraic and transcendental extension fields. Algebraic extension fields can be constructed as modular univariate polynomials (computing modulo an irreducible polynomial) and are as such computable fields. Transcendental extension fields can be constructed as (multivariate) polynomial fractions (also called rational functions) and as we can efficiently compute multivariate polynomial greatest common divisors to remove common factors, these fields are also computable. Sub-fields of algebraic extension fields, like real algebraic and complex algebraic extension fields are also of great importance and are also computable fields. The design and the implementation of these field extensions together with the modeling of the relation between algebraic and real algebraic extension fields is the topic of this paper.

When trying to implement our research agenda in a given programming language (Java) – in contrast of inventing a suitable language [2] – some design problems will eventually remain. We give a summary of such remaining design problems or implementation trade-offs and show how to solve them in Scala [3].

1.1 Related work

The related work published on type systems for computer algebra or abstract data type (ADT) approaches to computer algebra has been summarized in [1]. Type-safe design considerations in computer algebra are mostly centered around the *axiom* computer algebra system and are described, e.g. starting with [4, 2]. In application areas like constructive algebraic topology there exists strong demand for type-safe algorithms. For example the Kenzo system takes an object oriented approach with strong run-time type system [5]. The system handles 'chain complexes', which consist of algebraic structures together with mappings between them. Also for constructions in the formal theory of differential equations there is demand for object oriented software, see e.g. [6]. MuPAD has a simple object oriented layer for algebraic structures and so called categories, see [7]. DoCon has support for field extension towers in a Haskell package [8]. Using a suitable existing programming language has restrictions compared to the Axiom approach [2]. However, there are also benefits: we do not have to invent and improve memory management, parallel computing and networking support [9] and can ride on the advantages of computer science in this area [10]. Further related work is mentioned in the paper as required.

1.2 Outline

In Section 2 we discuss the design and implementation of algebraic and transcendental extension fields together with the modeling of real algebraic and complex algebraic extension fields. Section 3 presents a summary of design problems of our typed object-oriented approach when using Java as implementation language and presents possible solutions in Scala. Finally Section 4 draws some conclusions.

2 Algebraic structures as typed objects

In this section we first give an introduction into the object oriented type systems. For more details see our earlier articles [1, 11–13]. Then we discuss the design and implementation of algebraic and transcendental extension fields together with the modeling of real algebraic and complex algebraic extension fields. The constructed extension fields can be used in other algorithms on polynomials with coefficients from such fields, for example in (parallel) Gröbner base or greatest common divisor computations. Other algorithms like polynomial factorization will however require an implemented case for such fields. Due to space limitations we will not discuss performance, but will give some hints on computing times and possible improvements. We will also not be able to discuss mappings of elements between the various extension rings and fields, as for example evaluation homomorphisms between polynomial rings. Currently all such mappings and their application have to be coded explicitly, but some automatic mapping construction and convenient coercion would be desirable, at least in scripting interpreters (for a special case see Subsection 3.2).

2.1 Ring elements and ring factories

The basic building blocks of the type system consists of the interfaces `RingElem` and `RingFactory` and the classes which implement them, see figure 1. `RingElem` defines the methods which we expect to be available on all ring elements, for example `multiply()`, `isZERO()` or `isUnit()` with the obvious meanings. The construction of ring elements is done by factories, modeled after the *abstract factory* creational design pattern [14]. The factory `RingFactory` defines the construction methods for elements, for example `getONE()` to create the one element from the ring, `parse()` to create an element from a string representation or query methods such as `isAssociative()`.

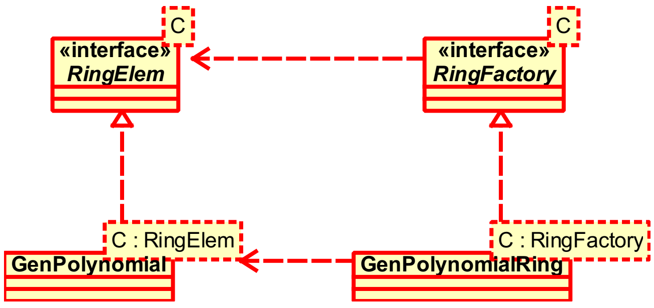

Fig. 1. Basic types

The polynomial class `GenPolynomial` with type parameter `C` for the coefficient type implements the `RingElem` interface and specifies that coefficients must be of type `RingElem`. In addition to the methods mandated by the interface, the `GenPolynomial` implements the methods like `leadingMonomial()` or `degree()`. Polynomials are to be created with a polynomial factory `GenPolynomialRing`. In addition to the ring factory methods it defines for example methods to create random polynomials. The constructor for `GenPolynomialRing` takes parameters for a factory for the coefficients, the number of variables, the names for the variables and a term order object `TermOrder`. The relation between the factory of the coefficients and the polynomial ring factory is modeled after the (constructor) dependency injection pattern and implements the inversion of control principle.

Figure 1 shows dependency arrows from the factories to the element interfaces as factories create the respective elements. The modeling of the constructors is not shown as it is not denotable in Java. The constructors of ring elements implement an opposite dependency : each constructor takes a corresponding ring factory as parameter. It is only indirectly enforced since the `RingElem` interface specifies a method `factory()` to obtain the corresponding factory.

The factory methods are not static (which is apparent from the modeling as an interface) since a ring factory might depend on other rings or specific parameters. In case of a polynomial factory it depends on a factory for the coefficients and at least the number of variables of the polynomial ring. By this modeling the types of elements of algebraic structures are not simply denoted in the program text but have to be created as programming objects (by instantiating the respective classes). Type denotations show up explicitly in Java program code and are mostly inferred in Scala via type resolution.

For example a polynomial $w^2 - 2 \in \mathbb{Q}[w]$ can be constructed by first constructing an object for the ring $\mathbb{Q}[w]$ and then reading and constructing the polynomial $w^2 - 2$ with the factory method `parse()`.

```
BigRational rf = new BigRational(1); // here element = factory
GenPolynomialRing<BigRational> pf
 = new GenPolynomialRing<BigRational>(rf,new String[]{ "w" });
GenPolynomial<BigRational> a = pf.parse("w^2 - 2");
```

This example is continued in Subsection 2.3.

2.2 Algorithms and factories

Our implemented algorithms are in fact meta-algorithms or functors. They do not only compute elements of algebraic structures but simultaneously construct the required algebraic structures during the computation. So several implemented methods map pairs of algebraic structures together with some elements to other algebraic structures and elements. For example Hensel lifting is an algorithm which maps

$$((\mathbb{Z}[x], a), (\mathbb{Z}_p[x], (a_1, ..., a_r)), (\mathbb{N}, k)) \mapsto (\mathbb{Z}_{p^k}[x], (b_1, ..., b_r)).$$

With the meaning $(a \in \mathbb{Z}[x], (a_1, ..., a_r) \in \mathbb{Z}_p[x]^r, k \in \mathbb{N}) \mapsto (b_1, ..., b_r) \in \mathbb{Z}_{p^k}[x]^r$. Using type annotations it would read

$$(a : \mathbb{Z}[x], (a_1, ..., a_r) : \mathbb{Z}_p[x]^r, k : \mathbb{N}) \mapsto (b_1, ..., b_r) : \mathbb{Z}_{p^k}[x]^r.$$

It is important to understand that $\mathbb{Z}_{p^k}[x]$ is an object constructed during the computation. The type annotation hides this fact. Note, the rings $\mathbb{Z}_p$ and $\mathbb{Z}_{p^k}$ are not distinguished by a Java type. To correctly model this as distinct types needs the concept of *dependent types* which is not available in Java 6 but is available in Scala, see Subsection 3.2.

2.3 Algebraic and transcendental field extensions

In this subsection we discuss the typed object oriented modeling of algebraic extension rings / fields, see figure 2. In slight abuse of number theoretic terminology we call elements of algebraic extensions *algebraic numbers*. The modeling of real and complex algebraic numbers is discussed in the next subsection (2.4). The construction of quotient fields of polynomial rings is not discussed further as it does not present new modeling challenges. For an introduction to algebraic fields and more references see Subsection 2.1 in [15].

Algebraic numbers are elements of some algebraic extension field L of a field K. If L is generated by a single element α we write $L = K(\alpha)$. L being an algebraic extension then means that there exists a polynomial $f \in K[x]$ such that $f(\alpha) = 0$ in L. If we can compute in K, the field L can be represented as $L = K[x]_{/(f)}$ and we can compute also in L. This representation is slightly ambiguous as it is not specified which conjugate of α is meant. With this representation L can be implemented as the residue class ring of $K[x]$ modulo the defining monic minimal polynomial $f \in K[x]$. It is implemented by class `AlgebraicNumberRing` which implements the interface `RingFactory` as mentioned before. Elements of this ring are implemented by class `AlgebraicNumber` implementing the interface `RingElem`. The constructor for `AlgebraicNumber-Ring` requires the defining polynomial to be provided. The name 'ring' is chosen, because the defining polynomial might not be irreducible. The constructor for `AlgebraicNumber` elements requires the corresponding algebraic number factory and an element $a \in K[x]$, which represents $a(\alpha) \in K(\alpha)$.

The construction of algebraic extension fields works uniformly for all fields K. For example it could be $\mathbb{Z}_p$, $\mathbb{Q}$, another algebraic extension $K(\beta)$ or a transcendental extension $K(y)$. Transcendental extensions are implemented by classes `QuotientRing` with elements `Quotient` from package `edu.jas.ufd`. By this design we can implement arbitrary towers of field extensions of some base field $\mathbb{Z}_p$ or the rational numbers $\mathbb{Q}$. For example, we can construct

$$\mathbb{Q}(\sqrt{2})(x)(\sqrt{x}) \text{ or } \mathbb{Z}_p(x)[y].$$

The latter denotes a polynomial ring over an infinite field of finite characteristic. For example the construction of the first field can be done step by step as in the following sequence.

$$\begin{aligned} \mathbb{Q} \mapsto_1 \mathbb{Q}[w] \mapsto_2 \mathbb{Q}[w]_{/(w^2-2)} &\mapsto_3 (\mathbb{Q}[w]_{/(w^2-2)})(x) \\ \mapsto_4 (\mathbb{Q}[w]_{/(w^2-2)})(x)[wx] &\mapsto_5 (\mathbb{Q}[w]_{/(w^2-2)})(x)[wx]_{/(wx^2-x)} \end{aligned}$$

Step 1 constructs a polynomial ring over the rational numbers, step 2 constructs an algebraic number ring with polynomial $w^2 - 2$ to represent $\mathbb{Q}(\sqrt{2})$. Step 3 constructs a polynomial ring in x and the quotient field of it, Step 4 constructs a polynomial ring for the ring from step 3 and finally step 5 constructs an algebraic number ring with polynomial $wx^2 - x$ to represent $\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})$. These steps can be followed exactly in the programming language with the construction of objects from the classes `GenPolynomialRing`, `AlgebraicNumberRing` and `QuotientRing`. Again, the construction is explicit and we can therefore build any such field extension towers as desired.

In the example, an element of $\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})$ is denoted as

```
AlgebraicNumber<Quotient<AlgebraicNumber<BigRational>>> elem;
```

When creating elements of these rings it is first necessary to construct the corresponding algebraic structures as programming objects in the same sequence as above. A sequence to create a factory for `elem` could be as follows.

```
... // see above
GenPolynomial<BigRational> a = pf.parse("w^2 - 2");
AlgebraicNumberRing<BigRational> af
 = new AlgebraicNumberRing<BigRational>(a);
String[] vx = new String[]{ "x" };
GenPolynomialRing<AlgebraicNumber<BigRational>> tf
```

```
 = new GenPolynomialRing<AlgebraicNumber<BigRational>>(af,vx);
QuotientRing<AlgebraicNumber<BigRational>> qf
 = new QuotientRing<AlgebraicNumber<BigRational>>(tf);
String[] vw = new String[]{ "wx" };
GenPolynomialRing<Quotient<AlgebraicNumber<BigRational>>> qaf
 = new GenPolynomialRing<Quotient<AlgebraicNumber<BigRational>>>(qf,vw);
GenPolynomial<Quotient<AlgebraicNumber<BigRational>>> b
 = qaf.parse("wx^2 - x");
AlgebraicNumberRing<Quotient<AlgebraicNumber<BigRational>>> fac
 = new AlgebraicNumberRing<Quotient<AlgebraicNumber<BigRational>>>(b);
```

The first field extension $\mathbb{Q}(\sqrt{2})$ is constructed as object in `af` as algebraic number ring. Then the transcendental extension $\mathbb{Q}(\sqrt{2})(x)$ is constructed as quotient field in object `qf`. And the last extension $\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})$ is constructed again as algebraic number field in object `fac`. Then it is possible to construct elements from this field, e.g. `elem = fac.parse("wx + x^5");`

The construction process looks tedious and contains a lot of 'boiler plate' type denotations. However, the construction is very precise, it is type safe and explicit! So every type and semantics of algebraic structures can be precisely constructed, an important advantage for the engineering of mathematical software libraries. Note, the type denotations are minimized in Scala through its type resolution capabilities, see Subsection 3.2.

Recall, that in Java the type information is 'erased' in the byte code and so it is not accessible at run-time. So using Java and (Java based) Scala classes in Jython [16] or JRuby [17] scripting interpreters access the raw objects and we have only run-time type safety, see Subsection 2.5. In any case the construction of algebraic structures is the same and so the construction of elements of such structures is run-time type safe, as the corresponding factories must be constructed explicitly.

The construction process can be shortened by a class implementing the *builder pattern* [14] to make it easier as follows.

```
RingFactory fac = ExtensionFieldBuilder
                  .baseField(new BigRational(1))
                  .algebraicExtension("w", "w^2 - 2")
                  .transcendentExtension("x")
                  .algebraicExtension("wx", "wx^2 - x")
                  .build();
```

There are several algorithms which can work with such field towers. For example in [1] we have shown that one can factor polynomials with coefficients from the ring $\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})$. Primitive elements for multiple algebraic extensions can be computed by methods `primitiveElement()` from class `PolyUtilApp`. Moreover, residue class rings modulo multivariate (prime) ideals can be used as extension rings. Note, the `build()` method is a perfect place to implement structural optimizations and simplifications of the field tower. For example moving algebraic extensions to the "bottom" of the tower and moving transcendental extensions to the "top" of the field tower, or replacing some algebraic extensions by primitive elements, or replacing the tower by a single multivariate residue class ring represented by a Gröbner base. Also type specialization techniques could be implemented in this method, see [18].

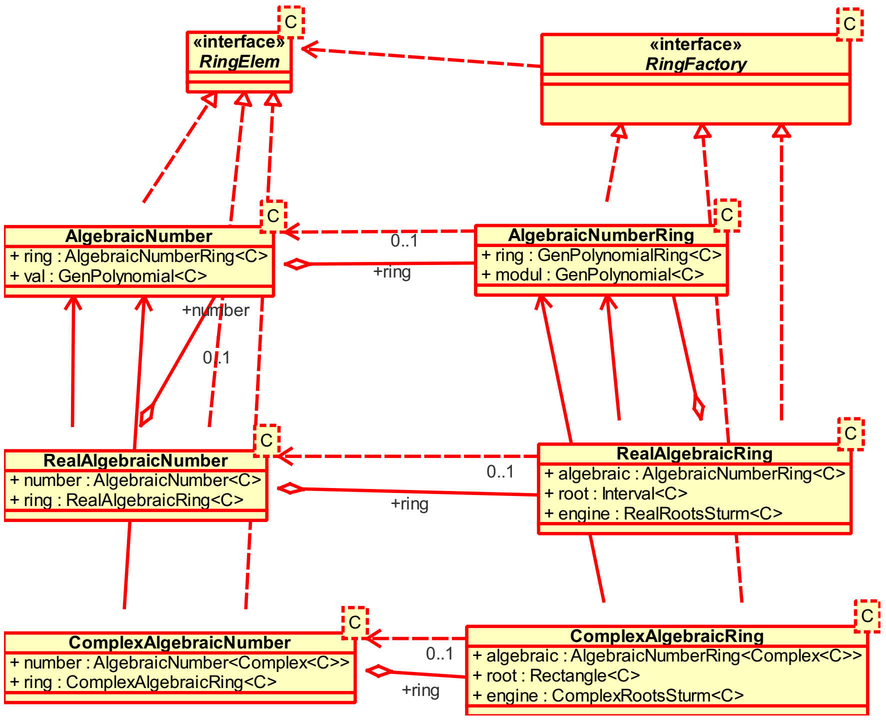
«interface»
RingElem
C
«interface»
RingFactory
C
AlgebraicNumber
+ ring : AlgebraicNumberRing<C>
+ val : GenPolynomial<C>
C
AlgebraicNumberRing
+ ring : GenPolynomialRing<C>
+ modul : GenPolynomial<C>
C
0..1
+ring
+number
0..1
RealAlgebraicNumber
+ number : AlgebraicNumber<C>
+ ring : RealAlgebraicRing<C>
C
RealAlgebraicRing
+ algebraic : AlgebraicNumberRing<C>
+ root : Interval<C>
+ engine : RealRootsSturm<C>
C
0..1
+ring
ComplexAlgebraicNumber
+ number : AlgebraicNumber<Complex<C>>
+ ring : ComplexAlgebraicRing<C>
C
ComplexAlgebraicRing
+ algebraic : AlgebraicNumberRing<Complex<C>>
+ root : Rectangle<C>
+ engine : ComplexRootsSturm<C>
C
0..1
+ring

Fig. 2. Algebraic Number classes

2.4 Real algebraic numbers and complex algebraic numbers

Real algebraic numbers $g(\alpha)$ are elements of a real algebraic field extension $L = K(\alpha)$ over a field K, for $\alpha \in \mathbb{R}$. They are represented as polynomials $g \in K[x]$ modulo a defining polynomial $f \in K[x]$ for the algebraic field extension together with an isolating interval $I = [l, r] \subseteq \mathbb{R}$ for a specific real root $\alpha \in I$ of f. The rationale for real algebraic numbers as programming objects is the ability to represent results of polynomial and ideal root finding algorithms precisely and not as unstructured isolating intervals, as it is done by contemporary commercial computer algebra systems. Note, $K(\alpha)$ is a sub-field of $E = K[x]_{/(f)}$. So one would like to implement $K(\alpha)$ as a sub-type of E via the sub-class implementation scheme.

However, to avoid the problem of type erasure for sub-classes with interfaces we model the relation according to the *delegation concept*, a compositional design concept [14]. That is, we do not model real and complex algebraic numbers as sub-classes of algebraic numbers but use delegation to algebraic numbers and algebraic number factories, see figure 2. The main advantages and disadvantages of delegation versus the inheritance approach are discussed in Subsection 3.1.

The implementation is contained in class `RealAlgebraicNumber` with factory class `RealAlgebraicRing`. The factory class contains an instance of the real root computation `engine`, which is used to refine intervals as required. The real root computation is contained in class `RealRootsSturm` for computation via Sturm sequences. The class is a sub-class of `RealRootAbstract`. There exist faster algorithms to compute isolating intervals than Sturm sequences, they will be implemented in future releases (see Subsection 2.1 in [15]). This design allows then the definition of polynomials with real algebraic coefficients

```
GenPolynomial<RealAlgebraicNumber<BigRational>>.
```

Moreover, for such polynomials we can also use real root isolation algorithms and instantiate and use for example

```
RealRootsSturm<RealAlgebraicNumber<BigRational>>.
```

This is possible since we implemented method `realSign()` which is used in the method `signum()` of a real algebraic number. `BigRational` and `Real-AlgebraicNumber` implement the interface `Rational` which defines a method `getRational()` to compute a rational approximation of the middle of the isolating interval to a prescribed accuracy.

Multiple real algebraic field extensions, for example by the third root of 3 and its square root and the fifth root of 2 $\mathbb{Q}(+\sqrt[3]{3})(+\sqrt{+\sqrt[3]{3}})(+\sqrt[5]{2})$ using $[1, 2]$ as isolating intervals, can be constructed as follows.

```
fac = ExtensionFieldBuilder
      .baseField(new BigRational())
      .realAlgebraicExtension("q", "q^3 - 3","[1,2]")
      .realAlgebraicExtension("w", "w^2 - q","[1,2]")
      .realAlgebraicExtension("s", "s^5 - 2","[1,2]")
      .build();
```

One possible implementation of complex algebraic numbers is similar to real algebraic numbers but using a bounding box in the complex plane to uniquely identify a specific complex algebraic number. Such a implementation uses the complex root computation from classes `ComplexRootsSturm` which is a subclass of `ComplexRootsAbstract`. Unfortunately this representation of complex algebraic numbers can not be used in a recursive setting since it is not possible to obtain a real algebraic representation for the real or imaginary parts from it. As a consequence `ComplexRootsSturm<ComplexAlgebraicNumber<.>>` can not be implemented. An alternate representation is as real roots of the ideal generated by the real and imaginary part of the given polynomial. That is, after substitution of $z \mapsto a + bi$ in the polynomial $f(z)$, we have $f(a, b) = f_r(a, b) + f_i(a, b)i$. Then we consider the real roots of the ideal generated by $f_r(a, b)$ and $f_i(a, b)$. So a specific $\gamma_k \in L = L'(i)$ with $f(\gamma_k) = 0$ is represented as $\alpha_k + \beta_k i \in L'(i)$ with $f_r(\alpha_k, \beta_k) = 0$ and $f_i(\alpha_k, \beta_k) = 0$ where $L' = K(\alpha, \beta)$ with real algebraic numbers α and β. All required algorithms are already implemented in classes `Ideal` and `PolyUtilApp` in package `edu.jas.application`. So we arrive at a representation as `Complex<RealAlgebraicNumber<RealAlgebraic-Number<.>>>` which is suitable in the recursion, i.e. for complex root computation of polynomials with coefficients from such rings.[3]

We conclude with the class `RootFactory`, see figure 3. It implements a functor for creating real algebraic numbers for polynomial real roots and a functor for creating complex algebraic numbers for polynomial complex roots. All functors internally construct first the real algebraic number rings respectively the complex algebraic number rings. The rings are accessible by the `factory()` method of the

[3] according to the fundamental theorem of algebra, for a constructive version see the Weierstraß-Durand-Kerner fixpoint method

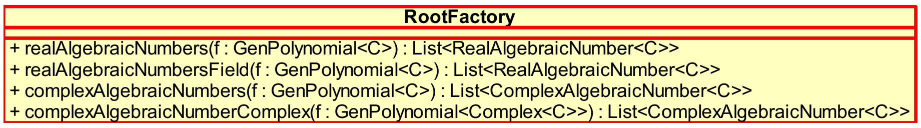
RootFactory
+ realAlgebraicNumbers(f : GenPolynomial<C>) : List<RealAlgebraicNumber<C>>
+ realAlgebraicNumbersField(f : GenPolynomial<C>) : List<RealAlgebraicNumber<C>>
+ complexAlgebraicNumbers(f : GenPolynomial<C>) : List<ComplexAlgebraicNumber<C>>
+ complexAlgebraicNumberComplex(f : GenPolynomial<Complex<C>>) : List<ComplexAlgebraicNumber<C>>

Fig. 3. Factory for algebraic numbers and fields

`RingElem` interface and an approximation of the magnitude can be obtained via the `getRational()` method. The respective methods `realAlgebraicRoots()` and `complexAlgebraicRoots()` for zero dimensional ideals are at the moment contained in class `PolyUtilApp` in package `edu.jas.application`.

For example we can compute real roots over the field `fac` using this root factory. Therefore we build a polynomial ring over `fac` in the variable, say `y`, parse for example the polynomial $y^2 - \sqrt{\sqrt[3]{3}\sqrt[5]{2}}$ and compute its two real roots.

```
GenPolynomial elem = pfac.parse("y^2 - w s");
List<RealAlgebraicNumber> roots = RootFactory.realAlgebraicNumbers(elem);
```

The real root isolation needs 1.2 seconds and the approximation to 50 digits needs a total of 5.2 seconds on an AMD running at 3 GHz and IcedTea6 JVM. The decimal approximation (via `getRational()` from the roots) shows the two real roots requested with 50 decimal digits as

```
-1.1745272686769866126436905900619307101229226521299
 1.1745272686769866126436905900619307101229226521299.
```

2.5 Algebraic structures in interactive scripting interpreters

The example $\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})$ from Subsection 2.3 can be constructed in a scripting interpreter in the same way as in the Java example (when not using `ExtensionFieldBuilder`). The (Jython) methods `AN()` and `RF()` construct algebraic respectively transcendental extensions and the `PolyRing` class represents a `GenPolynomialRing`. The `gens()` method for a ring R returns a list of generators as an R-algebra (including generators for coefficient rings represented in R). The generators are constructed in the sequence defined by the extension tower composition. `QQ` denotes the rational numbers.

```
Q = PolyRing(QQ(),"w2",PolyRing.lex);
[e,w2] = Q.gens();
root = w2**2 - 2;
Q2 = AN(root,field=True);
Qp = PolyRing(Q2,"x",PolyRing.lex);
Qr = RF(Qp);
Qwx = PolyRing(Qr,"wx",PolyRing.lex);
[ewx,wwx,ax,wx] = Qwx.gens();
rootx = wx**2 - ax;
Q2x = AN(rootx,field=True);
```

Finally a polynomial ring over this field extension can be constructed.

```
Yr = PolyRing(Q2x,"y",PolyRing.lex)
[e,w2,x,wx,y] = Yr.gens();
f = ( y**2 - x ) * ( y**2 - 2 ); // = y**4 - ( x + 2 ) * y**2 + 2 * x
```

Note, the variable `x`, set by the `gens()` method, correctly represents the generator `x` from ring `Qp` but as element of the polynomial ring `Yr`. There is a source of confusion with the method `gens()` as all returned generators must be listed in the assignment (and in the correct semantic sequence). So we also see the unused variables `e` (1 in `Yr`), `ewx` (1 in `Qwx`), `wwx` ($\sqrt{2}$ in `Qwx`), `ax` (x in `Qwx`) and `w2` ($\sqrt{2}$ in `Yr`). These usability problems can be solved in Scala as described in Subsection 3.2 or by an extension field builder for the scripting interface. For example with `EF(QQ()).extend("w2","w2^2 - 2").extend("x") .extend("wx","wx^2 - x").build()`.

In the example we can then compute the factorization over the extension fields (in 9.5 seconds on an AMD at 3 GHz, 5.7 seconds after JIT warm-up, also IcedTea6 JVM) and this looks as expected, `f = ( y - wx ) * ( y - w2 ) * ( y + wx ) * ( y + w2 )`.

Note, in scripting interpreters we only have run-time type safety, as the scripts are not compiled and statically type checked.

3 Problems

In this section, we summarize some problems we have studied in the library design and implementation. As we are using general purpose languages and not developing our own like in Axiom [2], there will inevitably exist some design problems which will have no satisfactory solution until said languages are made to evolve, if ever.

3.1 Generic types and Sub-classes

The first one can be demonstrated with the following JAS Java code defining the classes for polynomials

```
class GenPolynomial<C extends RingElem<C>>
      implements RingElem<GenPolynomial<C>> { ... }
```

and solvable polynomials

```
class GenSolvablePolynomial<C extends RingElem<C>>
      extends GenPolynomial<C> { ... }
```

The intention of this sub-class definition[4] is to be able to use algorithms written for polynomials also for solvable polynomials. One would like to add "`implements RingElem<GenSolvablePolynomial<C>>`" to the declaration of `GenSolvablePolynomial`. However this will lead to two usages of the `RingElem` interface with different type parameters. Due to the design of Java generics to be compatible with existing non-generic code, the nested type parameters are *erased* and the double interface usage would turn into a compile error.

The same problem occurs with classes `AlgebraicNumber` and `RealAlgebraicNumber` that we have studied in Subsection 2.4. If the latter could be made a subclass of the former, we could reuse algorithms, e.g. for primitive element computation. However, then we could not have `RealAlgebraicNumber<C extends`

[4] which should be reversed to be mathematically right, the problem here being that subclassing does not provide for subtyping in a rigorous sense, namely that of a subset of possible values

GcdRingElem<C>> to implement GcdRingElem<RealAlgebraicNumber<C>> but only GcdRingElem<AlgebraicNumber<C>>. This means that we could build polynomials over such a ring, but no further real algebraic number fields based on it, because for example no getRational() method is provided by the interface AlgebraicNumber<C>.

Thus we are conducted to resort to the delegation concept, which avoids the above type-erasure problem, but forbids to use algorithms written specifically for AlgebraicNumbers with RealAlgebraicNumbers.

A third solution, that we have investigated in ScAS, is to use neither delegation nor subclassing, but instead to have both classes to inherit from a common, abstract superclass[5]. That way, code reuse is made possible, and the problem of knowing what class should be a subtype/subset of the other, is avoided.

3.2 Dependent Types

A second problem is that in order to parametrize the list of variables of a polynomial or the module of integer residue classes, some dependent types are required (see [13], Subsection 7.3 "Dependent types"). We have investigated if we could use Scala's dependent types [19] and found that it is possible. In the rest of the subsection some familiarity with Scala is required.

The basic principle is illustrated below. Let us take modular integers for instance. We need to define a type like Mod(7) which depends on the value 7. The goal is to forbid arithmetic operations between integers with different moduli. In the current state of the library this is not done:

```
object Ring {
  trait Factory[T <: Ring[T]]
}
trait Ring[T <: Ring[T]] {
  val factory: Ring.Factory[T]
  def +(that: T): T
}
object Mod {
  def apply(mod: Int) = new Factory(mod)
  class Factory(val mod: Int) extends Ring.Factory[Mod] {
    def apply(value: Int) = new Mod(this)(value%mod)
  }
}
class Mod(val factory: Mod.Factory)(val value: Int)
         extends Ring[Mod] {
  def +(that: Mod) = factory(this.value+that.value)
  override def toString = value.toString
}
```

A use case is given below.

```
val r = Mod(7)
r(4)+r(4) // 1
val s = Mod(2)
r(4)+s(1) // problem : this works
```

[5] as explained in [13], Subsection 7.1 "Interfaces as types", such abstract class is roughly equivalent to a category in Axiom

So we have found a new design, where we have replaced the type parameter T by an abstract type member E, and now we can define the ring element as an inner class of the ring factory, or, in other words, as a type which depends on the factory.

```
trait Ring {
  type E <: Element
  trait Element {
    def +(that: E): E
  }
}
class Mod(val mod: Int) extends Ring {
  type E = Element
  class Element(val value: Int) extends super.Element {
    def +(that: E) = apply(this.value+that.value)
    override def toString = value.toString
  }
  def apply(value: Int) = new Element(value%mod)
}
```

A use case is given again below.

```
object r extends Mod(7)
r(4)+r(4) // 1
object s extends Mod(2)
r(4)+s(1) // type mismatch, as expected
```

In the case of polynomials we could have a dependent type which fulfills all our requirements as follows:

```
class Polynomial[C <: Ring, P](val ring: C,
                               val variables: Array[String],
                               val ordering: Comparator[P]) {
  type E // the type of the elements of the ring
}
```

When using Scala as a script interpreter [20], the design will moreover provide a solution to the problem mentioned in Subsection 2.5 that each definition of ring/extension field factory must redefine all generators in the factory tower. This will be achieved through implicit conversion, with such factory declarations as below, which will be able to "lift" values to the correct level in the ring/field tower.

```
implicit r extends Mod(7)
implicit p extends Polynomial(r, Array("x"))
implicit q extends Polynomial(p, Array("y"))
// and so on
```

We intend to rewrite the whole ScAS library based on this new principle. It will be the subject of a future publication.

3.3 Package structure

Lastly, there is a question about whether several algorithm flavors (for e.g. gcd, factorization and so on) could be implemented as polynomial (factory) subclasses as is currently investigated in ScAS, or should remain in distinct class hierarchies as in JAS (see [13], Subsection 7.5 "Recursive types").

4 Conclusions

We have discussed previously [1], how our typed object oriented approach with the Java and Scala programming languages makes it possible to implement non-trivial, type-safe algebraic structures which can be stacked and plugged together in unprecedented ways. In this paper we examined the modeling of field extensions using ring / field factories which represent the corresponding algebraic structures. All examples and the underlying mathematical algorithms from Section 2 (together with almost all algorithms from [21]) have been implemented and are available under a GPL license from a Git-repository at [11].

The construction process is very precise and explicit, so that no misinterpretation of the algebraic structure is possible. However, it is tedious and contains a lot of 'boiler plate' type denotations in case of Java as implementation language. The type denotations can be minimized in the Scala implementation through its type resolution capabilities. In scripting interpreters the type safety is only enforced at run-time due to the interpretative execution. But for the engineering of reliable and comprehensive mathematical software libraries the precise construction and type-safe modeling of algebraic structures is an important advantage. Compared to other computer algebra systems we can represent real roots not only by simple isolating intervals but as elements of algebraic structures which can be reused for further computations.

The design problems we encountered in Java can be resolved by modeling using alternative ways and in more advanced object oriented programming languages like Scala. Dependent types and the coercion facility in Scala will need further studies. Future work will include the implementation of faster algorithms for root isolation and to improve the (recursive) complex root isolation and representation.

Acknowledgments

We thank our colleagues Thomas Becker, Wolfgang K. Seiler, Thomas Sturm, Axel Kramer, Jaime Gutierrez, Sherm Ostrowsky, Ted Kosan and others for various discussions on the design of and the requirements for JAS and ScAS. Thanks also to the referees for the insightful suggestions to improve the paper.

References

1. Jolly, R., Kredel, H.: Generic, type-safe and object oriented computer algebra software. In: Proc. CASC 2010, Springer, LNCS 6244 (2010) 162–177
2. Watt, S.: Aldor. In: Computer Algebra Handbook, Springer. (2003) 265–270
3. Martin Odersky: The Scala programming language. Technical report, http://www.scala-lang.org/, accessed June 2011 (2003-2011)
4. Jenks, R., Sutor, R., eds.: axiom The Scientific Computation System. Springer (1992)
5. Rubio, J., Sergeraert, F.: Constructive algebraic topology. Bulletin des Sciences Mathematiques **126**(5) (2002) 389 – 412
6. Calmet, J., Seiler, W.M.: Computer algebra and field theories. Mathematics and Computers in Simulation **45** (1998) 33–37
7. Drescher, K.: MuPAD multi processing algebra data tool - Axioms, Categories and Domains. Technical report, http://www2.math.uni-paderborn.de/ (1995) Manuscript available via Citeseer.

8. Mechveliani, S.: DoCon - The Algebraic Domain Constructor. Technical report, http://botik.ru/pub/local/Mechveliani/docon/ (2007)
9. Maza, M.M., Stephenson, B., Watt, S.M., Xie, Y.: Multiprocessed parallelism support in ALDOR on SMPs and multicores. In: PASCO. (2007) 60–68
10. Taboada, G., Tourino, J., Doallo, R.: Java for high performance computing: Assessment of current research and practice. In: Proc. PPPJ'09, ACM (2009) 30–39
11. Kredel, H.: The Java algebra system (JAS). Technical report, http://krum.rz.uni-mannheim.de/jas/ (since 2000)
12. Kredel, H.: Evaluation of a Java computer algebra system. Lecture Notes in Artificial Intelligence **5081** (2008) 121–138
13. Kredel, H.: On a Java Computer Algebra System, its performance and applications. Science of Computer Programming **70**(2-3) (2008) 185–207
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Weseley (1995) Deutsch: Entwurfsmuster, Addison-Wesley, 1996.
15. Grabmaier, J., Kaltofen, E., Weispfenning, V., eds.: Computer Algebra Handbook. Springer (2003)
16. Jython Developers: Jython implementation of the high-level, dynamic, object-oriented language Python written in 100% pure Java. Technical report, http://www.jython.org/, accessed June 2011 (1997-2011)
17. JRuby Developers: JRuby a Java powered Ruby implementation. Technical report, http://jruby.org/, accessed June 2011 (2003-2011)
18. Dragan, L., Watt, S.: Type specialization in Aldor. In: Proc. CASC 2010, Springer, LNCS 6244 (2010) 73–84
19. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Proc. ECOOP'03, Springer LNCS (July 2003)
20. Jolly, R.: Object Scala found - a JSR223-compliant version of the Scala interpreter. In: Scala Days 2011. (2011) to appear
21. Becker, T., Weispfenning, V.: Gröbner Bases - A Computational Approach to Commutative Algebra. Springer, Graduate Texts in Mathematics (1993)